TRACES
Themes still Relevant At Current Software Engineering Studies

Ergebnisse des Bakkalaureatsseminars aus
Angewandte Informatik
Sommersemester 2007

Leitung:
Roland Mittermeir

Institut für Informatik-Systeme
Alpen-Adria Univerisität Klagenfurt

Klagenfurt, Juli 2007

INHALTSVERZEICHNIS

Vorbemerkungen

Roland Mittermeir
Institut für Informatik-Systeme
roland@isys.uni-klu.ac.at

Abstract

Dieser Sammelband umfasst die Endausarbeitungen des von mir im Sommersemester 2007 geleiteten Seminars aus Angewandter Informatik. Es stand unter dem Motto ***„TRACES: Themes still Relevant At Current Software Engineering Studies“****.*

Die im inhaltlichen Teil des Sammelbandes enthaltenen Arbeiten sind Bakkalaureatsarbeiten[1]*, die von Studierenden im 6. Semester ihres Informatikstudiums anzufertigen sind. Diese Vorbemerkungen sollen den die Organisationsform dieses Bakkalaureatsseminars und die Motivation für das Rahmenthema erläutern.*

1. Hintergrund

Das UG'02 sieht vor, dass Bakkalaureatsarbeiten im Rahmen von Lehrveranstaltungen“ abgefasst werden [1]. Im Bakkalaureatsstudium Informatik sind demgemäß je eine Bakkalaureatsarbeit im Seminar aus Angewandte Informatik und eine im Softwarepraktikum anzufertigen. Dabei hat, dem Lehrveranstaltungstyp entsprechend, die erste Arbeit theoretisch-konzeptioneller Natur zu sein. Sie wird in der Regel als Einzelarbeit verfasst. Die Praktikumsarbeit ist demgegenüber praktisch-anwengungsbezogen und wird in der Regel in Kleingruppen als Anwendungsprojekt erstellt [2].

Im Sommersemester 2007 fanden zwei parallel gehaltene Bakkalaureatsseminare statt. Die hier wiedergegebenen Arbeiten hatten das Ziel, klassische Aufsätze aufzuarbeiten und den Bezug der inzwischen als historisch einzustufenden Ideen zu aktuellen Themen der Informatik, speziell des Software-Engineering, herzustellen.

Wie diese Verbindung zwischen einer klassischen Referenz und aktuellen Themen der Informatik hergestellt wird, lag völlig in den Händen der Studierenden. Leser können sich überzeugen, dass diese sehr unterschiedliche Wege wählten. Allerdings möchte ich bei aller Varianz der Qualität der nachfolgenden Arbeiten doch festhalten, dass die meisten Studierenden sich der Aufgabe durchaus sehr gut entledigt haben und sich redlich Mühe gaben, wirklich gute Arbeiten abzugeben. Dass Einzelne auch einen flacheren Weg als ausreichend steil betrachteten und andere die Stufen des Verfahrens vorerst als nicht hinreichend ernst zu nehmende Hürden aufgefasst haben mögen, liegt in der Natur von Lehrveranstaltungen. Die zugehörigen Arbeiten zeigen noch deutliches Verbesserungspotential. Doch selbst in sehr guten Arbeiten werden noch einige Flüchtigkeitsfehler störend wirken.

Zwischen der Rolle eines Editors dieser Sammlung und der eines Prüfers schwankend, entschloss ich mich schließlich für letztere und übernahm die Endfassungen der abgegebenen Arbeiten ohne jeglichen editoriellen Eingriff. Schließlich sind es Prüfungsarbeiten und als solche halt noch mit Schwächen unterschiedlichen Ausmaßes behaftet.

In den folgenden Abschnitten dieses Vorwortes wird die Organisationsform des Bakkalaureatsseminars beschrieben und der Hintergrund des Rahmenthemas *„TRACES: Themes still Relevant At Current Software Engineering Studies“* erläutert. Das Vorwort schließt mit einigen didaktischen Reflexionen.

2. Organisationsform des Seminars

Das Seminar wurde, einem inzwischen bereits zur Tradition gewordenen Stil des Seminarleiters entsprechend, als Variation eines Konferenzseminars abgehalten.

In der Grundform dieser Seminarform wählen die Studierenden innerhalb eines Rahmenthemas selbständig ein Thema samt zugehöriger Literatur. Hier war durch die Vorgabe von Literatur, die für weitere Entwicklungen richtungweisend war, zwar eine Menge

[1]) 14 der 15 Arbeiten sind Bakkalaureatsarbeiten. Eine (*Are goto's don't do's?*) ist die „normale“ Seminararbeit eines MA-Studenten. Eine Studierende musste aus externen Gründen knapp vor Ende der Veranstaltung abbrechen und hat keine verbesserte Fassung abgegeben. Diese Arbeit fehlt mithin im Sammelband.

von klaren Startpunkten definiert, aber mit über 40 Arbeiten für etwa 20 erwartete Interessenten umfasste die Ausgangsmenge doch ein deutliches Überangebot. Daher sollten wenigstens früh Entschlossene ein den persönlichen Interessen nahe liegendes Thema finden können. Die Ausgangsarbeiten waren in der e-Learning Plattform MOODLE abgelegt und konnten daher von allen Seminaristinnen und Seminaristen angelesen werden.

Innerhalb der ersten zwei Wochen des Semesters konnte man im MOODLE Diskussionsforum eine Nachricht posten, welchen Basisartikel (in einigen Fällen handelte es sich um Paare zusammengehöriger Artikel) und damit welches *Thema* man gewählt hat. Diese Postings waren allen Seminarteilnehmern ersichtlich. Weiters markierte der Seminarleiter periodisch jene Arbeiten, die bereits gewählt und damit „vergeben“ waren.

Nach weiteren drei Wochen (also etwas über ein Monat nach Veranstaltungsbeginn) war ein *kommentiertes Inhaltsverzeichnis* abzugeben. Dieses sollte neben Titel und kurz erläuterter Gliederung ein Abstract enthalten, aus dem hervorging, wie man den Basisartikel in der Informatik-Gegenwart positionieren möchte. Die meisten (und besseren) dieser Ersteinreichungen gaben – wohl der Tradition meiner bisherigen Anforderungen an Extended Abstracts folgend – auch einige weitere Quellen an, die sie in die Ausarbeitung ihrer Arbeit einbeziehen wollten. Doch dies war nicht explizit gefordert (es sollte allerdings für künftige Auflagen dieses Modells, wie in der Ausgangsform dieser Veranstaltung praktiziert, gefordert werden).

Mit dieser Ersteinreichung begann ein wesentlicher neuer Abschnitt der Lehrveranstaltung: Ein erster *Peer Review*. Jeder Einreichung wurden 4 Reviewer zugewiesen. Drei davon stammten aus dem Kreis der Seminaristen, einer war standardmäßig der Lehrveranstaltungsleiter. Hiezu mag Skepsis auftreten. Man könnte meinen, dass sich einerseits Studierende so wenig kritisieren werden, wie eine Krähe einer anderen …, und außerdem läge zu diesem Zeitpunkt ja noch kaum Begutachtbares vor.

Beide Bedenken dürfen als falsch verworfen werden. Auch aus den in der Regel eine Seite umfassenden Ersteinreichungen ist nicht nur für geübte Reviewer sondern auch für Studenten, die bereit sind, sich dieser Mühe zu unterziehen, ersichtlich, ob hinter dem knappen Text bereits Substanz steht oder ob nur rasch etwas einigermaßen plausibel erscheinendes abgegeben wurde. Freilich variierten diese Einreichungen in Qualität und Umfang stark. Doch ein „zu viel“ ist an dieser Stelle ebenso unpassend wie ein „zu dünn“. Der Begutachtungsprozess wurde mit dem open source Produkt MyReview [3] durchgeführt. Für diese Begutachtung stand ein Zeitraum von 14 Tagen zur Verfügung. Dass bereits dieser erste Schritt ein Filter war, mag daraus ersehen werden, dass von insgesamt 27 Themeneinreichungen nur mehr 25 Studierende eine Kurzfassung abgaben und am Reviewprozess teilnahmen.

In der nächsten Phase (nach den Osterferien) waren *Kurzvorträge* in der Dauer von 8 Minuten vorgesehen. In diesen Vorträgen sollten die Studierenden einerseits den Ausgangsartikel ihrer Arbeit in knappen Worten vorstellen und anderseits zeigen, welche Stoßrichtung sie in ihrer eigenen Arbeit verfolgen wollen. An jeden dieser Vorträge schloss sich eine kurze Diskussion. Drei Studierende meldeten sich noch vor den Kurzvorträgen ab, sodass insgesamt 22 Vorträge an drei Seminartagen gehalten wurden.

Nächster Schritt war die Abgabe des *Vollbeitrags*. Die Abgabetermine waren entsprechend den Kurzvortragsterminen gestaffelt (Mitte bis Ende Mai). Die Abgabe erfolgte wiederum in MyReview und wurde wieder von 4 Personen *begutachtet*. Diese sollten mit den Gutachtern der Ersteinreichung ident sein. Durch die inzwischen eingetretenen Ausfälle (und durch wietere Ausfälle bei der Endabgabe) mussten jedoch einige Neuzuordnungen getroffen werden. Letztlich wurden nur mehr 17 Arbeiten eingereicht. Eine dieser Arbeiten wurde im Begutachtungsprozess ausgeschieden. Eine andere Arbeit kam trotz sehr positiver Begutachtung aus persönlichen Gründen der Studierenden nicht zum letzten verpflichtenden Schritt, dem Vortrag.

Somit fanden im Zeitraum 6. bis 26. Juni insgesamt 15 *Vorträge* zu je 30 Minuten mit anschließend etwa 15 Minuten Diskussion statt. Die Studierenden hatten nach dem Vortrag noch bis 15. Juli (bewusst ein Termin in den Ferien) Zeit, ihren Vollbeitrag im Lichte der Ergebnisse des Begutachtungsverfahrens sowie auf der Grundlage der Diskussion nach ihrem Vortrag zu überarbeiten. Die in den folgenden Kapiteln in der Reihenfolge der Vorträge wiedergegebenen Arbeiten sind die Ergebnisse dieser Überarbeitung. Alle fünfzehn Vortragenden haben (einige mussten wohl) von der Möglichkeit einer Überarbeitung Gebrauch gemacht.

Als wesentliches Zusatz-Feature dieses Seminars ist noch die Kooperation mit dem SchreibCenter[2] der Universität Klagenfurt zu nennen. Die Seminarteilnehmer hatten die Möglichkeit, Teile ihrer Arbeit von Mitarbeiterinnen des SchreibCenters auf sprachliche Verbesserungsmöglichkeiten prüfen zu lassen. Einige haben diese Möglichkeit schon vor Abgabe der Erstfassung dieser Arbeit in Anspruch genommen. Andere

---

[2]) http://www.uni-klu.ac.at/uniklu/org/oe.jsp?orgkey=706

bekamen die Konsultierung des SprachCenters im Rahmen des Begutachtungsprozesses empfohlen.

3. Motivation des Inhalts

Für Lehrende stellt sich das didaktische Problem, der Generation der jetzt etwa Zwanzigjährigen jene Konzepte der Informatik näher zu bringen, die aus einer Laptop- und Java-Perspektive als im doppelten Sinn des Wortes *überkommen* erscheinen müssen, auch wenn diese Arbeiten auf der Basis einer breiteren Perspektive als grundlegend einzustufen sind.[3]

Es stellte sich daher für mich die Frage, ob und wie weit Studierende in der Lage sind, aus eigener Einschätzung aktueller Entwicklungen des eigenen Fachs den Stellenwert derartiger Arbeiten zu erkennen und eine Brücke zur Gegenwart zu bauen. Im günstigsten Fall sollten diese Brücken, so sie tragen, auch als Motivationshilfen an jenen Stellen dienen, an denen man in anderen Lehrveranstaltungen kurz auf die Wurzeln aktueller Entwicklungen eingeht.

Da das Bakkalaureatsseminar keine spezifische fachliche Umschreibung hat, bot es die Chance, obige Fragestellung experimentell zu behandeln. Dazu wurden noch in den Semesterferien die in Tabelle 1 angeführten Arbeiten in die eLearning Plattform MOODLE gestellt, damit sich Studierende rechtzeitig eine Arbeit und damit ein Themengebiet wählen, das ihren Interessen entspricht. Um entsprechende Freiheitsgrade zu sichern, wurden in diese Sammlung etwa doppelt so viel Arbeiten aufgenommen, als Teilnehmer in der Lehrveranstaltung erwartet wurden.

Die Auswahl selbst hat einen Bias in Richtung Programmierung und Software Engineering. Dies entspricht dem Fachgebiet des Seminarleiters. Doch da das Bakkalaureatsseminar als Seminar aus Angewandte Informatik ausgeschrieben ist, wurden auch Arbeiten aus den Bereichen Datenbanken, Knowledge Engineering und Betriebssysteme aufgenommen. Neben fachlichen Argumenten spielte auch Verfügbarkeit und Lesbarkeit eine Rolle. Daraus ergab sich, dass ein weiterer Bias in Richtung auf Arbeiten, die auf acm- oder IEEE-Servern elektronisch verfügbar sind und für die via Universitätsbibliothek Lehrenden wie Studierenden der Zugang frei möglich ist, bevorzugt wurden. Bedauerlicherweise schieden nach diesen Kriterien einige Arbeiten aus, die ich sonst sehr gerne aufgenommen hätte.

Die gewählten Arbeiten sind in Tab. 1 gelb (Basisartikel, die gewählt, aber nicht ausgearbeitet wurden) oder grün (Artikel, für die eine Vollversion ausgearbeitet wurde) unterlegt[4]. In Kenntnis des Curriculums darf ich festhalten, dass hierbei nicht nur Arbeiten gewählt wurden, die in einzelnen Lehrveranstaltungen zweifellos schon Erwähnung fanden, sondern Thema oder Neugierde auch zu einigen Arbeiten greifen ließ, die subjektives Neuland sein sollten. Ebenso ist davon auszugehen, dass man die Namen vieler in Tabelle 1 genannter Autoren schon einmal gehört hat. Aber sicherlich nicht alle. Dennoch wurden auch von den subjektiv unbekannten Autoren Arbeiten gewählt.

Blickt man in die Ausarbeitungen der Studierenden, zeigt sich, dass diese – wenig überraschend – vieles auf Themen lenken, denen innerhalb des Bakkalaureatsstudiums angemessener Raum gewidmet wurde. Es zeigt sich jedoch auch, dass sich einige der Mühe unterzogen, in neue Gebiete einzudringen.

Im Sinn des Experiments nahm die Seminarleitung möglichst wenig Einfluss darauf, wie Studierende den Wert der Ausgangsarbeit einschätzten und wohin sie die Brücke in die Gegenwart bauten. In einigen wenigen Fällen wurde aber im Rahmen der Begutachtung stärkere Fokussierung eingemahnt, damit letztlich eine innerhalb des vorgegebenen Seitenumfangs sinnvolle Darstellung erzielt werden kann. Wenn dadurch in einigen Fällen recht dünne Brücken entstanden und in anderen vielleicht der Eindruck einer gewissen Schieflage eingemahnt werden könnte, wäre solche Kritik zwar objektiv berechtigt, sie würde aber gegen die selbst auferlegten Regeln des Seminars verstoßen.

Ebenso könnte man kritisch anmerken, dass zu den historischen Arbeiten eigentlich auch der historische Hintergrund, auf dem diese Arbeiten ursprünglich geschrieben wurden, aufgearbeitet werden sollte. Dies fehlt in den schriftlichen Arbeiten weitgehend und wurde in den Vorträgen bzw. wenn, dann eher in den Diskussionen, nur sporadisch angesprochen. Allerdings geht diese Kritik ebenfalls von einer anderen Zielsetzung der Lehrveranstaltung aus. Sie ist daher nicht gegen die Arbeiten sondern allenfalls gegen die Zielrichtung des Seminars zu wenden.

4. Erfahrungen

Aus meiner Sicht hat das Konzept des Seminars inhaltlich wie organisatorisch gut gegriffen. Dass sich das Teilnehmerfeld von 27 Studierenden, die sich an einem Thema versuchten, relativ bald auf 22 reduzierte, war zu erwarten. Ich ging vielmehr aufgrund des mit dieser Organisationsform verbundenen Aufwands aller Beteiligten bereits vor Beginn der Kurz-

[3]) Eine umfassendere Darstellung der Motivation des Rahmenthemas findet sich in [4].

[4]) Wenn Zeilen in der ersten Spalte abgesehen vom 3. Buchstaben eine identische Kennung aufweisen, bestand das zugehörige Basisthema aus jeweils zwei Artikeln.

vorträge davon aus, dass letztlich nicht mehr als zwanzig Personen übrig bleiben würden. Alles andere hätte auch zu Problemen in der Abwicklung der Hauptvorträge geführt. Dass letztlich nur 17 Vollbeiträge ausgearbeitet wurden, von denen 15 hier abgedruckt sind, ist allerdings doch bedauerlich.

Methodisch hat das enge Zeitraster mit vielen Interventionspunkten den Vorteil, dass, im Unterschied zu klassischen Seminarformen, Studierende gehalten sind, sich relativ kontinuierlich über ein Semester hinweg mit der Thematik ihrer Seminararbeit zu beschäftigen. Optimistische Last-Minute-Aktionen, die sonst zu oft, aber stets ohne Erfolg, versucht werden, sind bei einem Konferenzseminar weitestgehend ausgeschlossen. Freilich ist dieser Vorteil nur relativ. Drei Ersteinreichungen von Vollbeiträgen waren noch so skizzenhaft, dass sie einen strengen Review-Prozess kaum überlebt hätten. Um welche es sich dabei handelt, ist wohl auch den überarbeiteten Endversionen noch anzusehen.

Als weiteres wesentliches Element dieser Seminarform sehe ich, unabhängig vom gewählten Seminarthema, den Prozess der Peer-Reviews. Dieser erlaubt den Studierenden, Feedback von mehr Personen als bloß der Seminarleitung zu bekommen. Gerade das Feedback von Peers, das in vielen Fällen qualitativ sehr hochstehend war, wirkt stark.[5] Weiters bietet der Peer-Review Studierenden die Möglichkeit, sich im Verfassen konstruktiver Kritik zu üben. Aber er bietet den Gutachtern natürlich auch die Chance, zu sehen, wie sich andere einer vergleichbaren Aufgabe stellten. Wünschenswert wäre hierbei natürlich, dass jedes Gutachter-Panel und jedes Paket aus zu begutachtenden Arbeiten ein breites qualitatives Spektrum abdeckt. Dies lässt sich vorab nicht realisieren. Bei der 3+1 Zusammenstellung ist die Chance auf eine gewisse Varianz innerhalb der Qualität der zu begutachtenden Arbeiten wie auch innerhalb der Qualität der erhaltenen Arbeiten aber doch relativ hoch.

Bleibt als letztes Element die Qualität der Lehrveranstaltung als solcher. Diese lebt natürlich von der Motivation der Teilnehmerinnen und Teilnehmer. Wenn allerdings bei den einzelnen Vorträgen mindestens 4 Personen im Raum sind, die sich mit der vorgetragenen Arbeit schon auseinander gesetzt haben, wird die Diskussion wohl besser sein, als wenn Vortragende und Zuhörer ihre physische Präsenz nur unter das Motto stellen, nur ja nichts zu fragen oder zu sagen, weil man im Gegenzug ja auch gefragt werden könnte. Hier war die Diskussionszeit jedoch vor allem bei den Kurzvorträgen aufgrund der in dieser Phase noch hohen Teilnehmerzahl eigentlich zu kurz. In vermindertem Maße galt das auch bei einigen Hauptvorträgen.

Somit bleibt mir nur, mich bei den Teilnehmerinnen und Teilnehmern für deren Engagement und für die Bereitschaft, Arbeiten und Referate von Kollegen konstruktiv zu kritisieren zu bedanken und Ihnen bei der Lektüre der folgenden Beiträge einige spannende Einblicke zu wünschen.

10. References

[1] Bundesgesetz über die Organisation der Universitäten und ihre Studien (Universitätsgesetz 2002), BGBl. I, Nr. 120/2002, Fassung BGBl. I Nr. 74/2006, § 51, Abs. 2, lit. 7 sowie § 124, Abs. 10.

[2] o.V.: Studienplan für das Bakkalaureatsstudium und Magisterstudium Informatik an der Universität Klagenfurt, Mitteilungsblatt, Univ. Klagenfurt, 26. Juni 2003.

[3] MyReview, http://myreview.lri.fr/, last accessed, July 2007.

[4] R. Mittermeir: The Challenge of Teaching Foundational Ideas in a Modern Informatics Curriculum; Proc. MEDICHI 2007, Methodic and Didactic Challenges of the History of Informatics, OCG 2007, pp. 143 – 150.

Danksagung

Besonderer Dank gilt den Kolleginnen des Schreib-Centers. Unter Leitung von Prof. Ursula Doleschal und Mag. Carmen Mertlitsch haben studentische MitarbeiterInnen dieser Serviceeinrichtung der Universität jenen Seminaristen, die dieses Service in Anspruch nahmen, Hinweise zur stilistischen und sprachlichen Verbesserung ihrer Arbeit gegeben.

[5]) Ob Feedback von Peers oder von der Seminarleitung kam, war den Empfängern nicht ersichtlich. In manchen Fällen mag man aus dem Inhalt Vermutungen geschöpft haben. In der Regel lagen war dies allerdings nicht möglich und ich bin auf Grund des Inhalts mancher studentischer Gutachten überzeugt, dass einige derartiger Vermutungen falsch waren.

Tabelle 1

TRACES

Themes still Relevant At Current software Engineering Studies

Thema	Autor	Titel	Quelle	ge-wählt	ausge-arbeitet
	Programming and Programming Languages				
p1	C. Böhm, G. Jacopini	Flow Diagrams, Turing Machines and Languages with Only Two Fromation Rules	Comm. ACM, Vol 9 (5), May 1966, pp. 366-377		
p2a	E. Dijkstra	Go To Statement Considered Harmful (Letter to the Editor)	Comm. ACM, Vol 11 (3), March 1968, pp. 147 -148	x	x
p2b	D. Knuth	Structured Programming with go to Statements	in R.T. Yeh (ed) Current Trends in Programming Methdolology, Vol. 1, 1977, pp. 140-194	x	x
p3	N. Wirth	On the Composition of Well-Structured Programs	acm Computing Surveys, Vol. 6 (4), Dec 1974, pp. 247-259	x	x
p4	N. Wirth	Program Development by Stepwise Refinement	Comm. ACM, April 1971, Vol. 14 (4), pp. 221-227	x	
p5a	M. Shaw, W.A. Wulf, R.L. London	Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators	Comm. ACM, Vol 20 (8), Aug 1977, pp. 553 - 564		
p5b	B. Liskov, A. Snyder, R. Atkinson, C. Schaffert	Abstraction Mechanisms in CLU	Comm. ACM, Vol. 20 (8), Aug. 1977, pp. 564-576		
p6a	D.L. Parnas	On the Criteria to Be Used in Decompsing Systems into Modules	Comm. ACM, Vol. 15 (12), Dec. 1972, pp. 1053-1058		
p6b	D.L. Parnas	A Technique for Software Module Specification with Examples	Comm. ACM, Vol. 15(5), May 1972, pp. 330-336		
p7	J. Backus	Can Programming Be Liberated from the von Neumann Style? A Functional Style and Ist Algebra of Programs	Comm. ACM, Vol. 21 (8), pp. 613-641		
	Specification				
s1	C.A.R. Hoare	Communicating Sequential Processes	Comm. ACM, Vol. 21 (8), August 1978, pp 666-677		
s2	B.H. Liskov, S.N. Zilles	Specification Techniques for Data Abstractions	in R.T. Yeh (ed) Current Trends in Programming Methdolology, Vol. 1, 1977, pp 1-32	x	x
s3a	D. Harel, E. Gery	Executable Object modeling with Statecharts	IEEE Computer, July 1979, pp. 31-42	x	x
s3b	D. Harel, E. Gery	Executable object modeling with statecharts	Proc. 18th ICSE, March 1996, pp. 246 ff.	x	x
	Software Technology				
t1a	M. Weiser	Program slicing	Proc. 5th ICSE, 1981, San Diego, pp. 439-449	x	
t1b	M. Weiser	Programmers use slices when debugging	Comm. ACM, Vol. 25 (7), pp 446-452	x	
t2	F. DeRemer, H.H. Kron	Programming-in-the-Large vs. Programming-in-the-Small	Proc.Internat. Conference on Reliable Software, Los Angeles, 1975, pp. 114-121	x	
t3	W.P. Stevens, G.J. Myers, L.L. Constantine	Structured Design	IBM Systems Journal, May 1974, pp. 115-139	x	
7	D.T. Ross, K.E. Schoman Jr.	Structured Analysis for Requirements Definition	IEEE Trans. on Software Eng. Vol. SE-3, Jan 1977, pp. 6-15	x	
t5	M.E. Fagan	Design and Code Inspections to Reduce Errors in Program Development	IBM Systems Journal, Vol 15(3),. 1976, pp.182-211	x	x
t6	B.W. Kernighan, J.R. Mashey	The Unix Programming Environment	IEEE Computer, April 1981, pp. 12-22	x	x

Software Development Process

h1	B.W. Boehm	Software Engineering	IEEE Trans on Computers, Vol. C-25 (12), Dec. 1976, pp. 1226 - 1241	x	x
h2	Matsumoto	A Software Factory: An Overall Approach to Software Production	in: P. Freeman, Tutorial Software Reusability, IEEE, 1987, pp. 155-178	x	x
h3	R. Prieto-Diaz, P. Freeman	Classifying Software for Reusability	IEEE Software, March 1985, pp. 24-33	x	x
h4	P. Wegner	Capital Intensive Software Technology	IEEE Software, Vol. 1 (3), July 1984, pp. 7-45	x	x
h5	L. Osterweil	Software Processes are Software, Too	Proc. ICSE 9, Monterey, 1987, pp. 2-13.	x	x
h6	F.P. Brooks, Jr	The Mythical Man-Month	Datamation, Dec. 1974	x	
h7	L.A. Belady, M.M. Lehman	A Model of Large Program Development	IBM Systems Journal, No 3, 1976, pp. 225-252		
h8	V.R. Basili, R.W. Selby, D.H. Hutchens	Experimentation in Software Engineering	IEEE Trans on Software Eng. Vol. SE-12 (7), July 1986, pp. 733-743	x	
h9	L. H. Putnam	A General Empirical Solution to the Macro Software Sizing and Estimating Problem	IEEE Trans on Software Eng. July 1978		
h10	A.J. Albrecht, J.E. Gaffney Jr	Software function, source lines of code and development effort prediction: A software science validation	IEEE Trans on Software Eng. Vol. SE-9 (6), Nov. 1983, pp. 639-649		
h11	B. Curtis, S.B. Sheppard, P.M. Milliman, A.Borst, T. Love	Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics	IEEE Trans. on Software Eng. Vol. SE-5 (2), March 1979, pp. 95-104		

Data Bases and Information Systems

d1	E.F. Codd	A Relational Model of Data for Large Shared Data Banks	Comm ACM, Vol. 13 (6), June 1970, pp. 337-387		
d2	E.F. Codd	Extending the Database Relational Model to Capture More Meaning	ACM TODS, Vol 4 (4), Tec. 1979, pp. 397-434		
d3	P. P.-S. Chen	The Entity-Relationship Model - Toward a Unified View of Data	ACM TODS, Vol. 1 (1), March 1976, pp. 9 - 36		
d4	J.M. Smith, D.C.P. Smith	Database abstractions: aggregation and generalization	ACM TODS, Vol. 2 (2), June 1977, pp. 105-133	x	
d5	J. Mylopoulos, P.A. Bernstein, H.K.T. Wong	A Language Facility for Designing Database-Intensive Applications	ACM TODS, Vol. 5 (2), June 1980, pp. 185-207	x	
d6	S.J. Greenspan, J. Mylopoulos, A. Borgida	Capturing More World Knowledge in the Requirements Specification	Proc. ICSE 1982, pp. 225-234	x	x
d7	Arango	Domain Analysis - From Art Form to Engineering Discipline	Proc 5th IWSSD, 1989, pp. 125-135	x	x

Knowledge Representation

a1	R.J. Brachmann	What IS-A Is and Isn't: "An Analysis of Taxonomic Links in Semantic Networks"	IEEE Computer, Vol 16 (10), Oct 1983, pp. 30-36	x	
a2a	L.A. Zadeh	Commonsense knowledge representation based on fuzzy logic	IEEE Computer, Vol 16 (10), Oct 1983, pp. 61-65	x	x
a2b	L.A. Zadeh	Knowledge Representation in Fuzzy Logic	IEEE Trans. on Knowledge and Data Engineering, Vol 1 (1), March 1989, pp. 89-100	x	x
a3	L.A. Zadeh	Fuzzy Logic = Computing with Words	IEEE Trans. on Fuzzy Systems, Vol 4 (2), May 1996, pp. 103-111	x	x

Operating Systems

o1	E.W. Dijkstra	The struchture of "THE"-multiprogramming system	Comm. ACM, Vol. 11 (5), May 1968, pp. 341-346	x	x
o2	D.M. Ritchie, K. Thompson	The Unix Time-Sharing System	Comm. ACM July 1974, pp. 365-375	x	

Seminar aus Angewandte Informatik
R. Mittermeir
Wintersemester 2006/07

Vorbesprechung und Anmeldung
Dienstag, 6. März, 8:15-9:45
SR 2.42

Bakkalaureatsseminar aus Angewandte Informatik
TRACES
Themes still Relevant At Current Software Engineering Studies
Aufruf zur Teilnahme und Seminarordnung

Das Bakkalaureatsseminar wird im Sommersemester 2007 in form eines e-Learning unterstützten gemeinsames Workshops geführt. Kooperation mit dem Schreibcenter wird angestrebt. Die diesbezüglichen Möglichkeiten hängen jedoch von der Teilnehmerzahl ab und können derzeit noch nicht fixiert werden.

Lehrziel ist, dass Studierende durch selbständige Bearbeitung eines aktuellen Themas aus Angewandter Informatik neben fachlichen Kenntnissen auch methodische Kenntnisse im Verfassen wissenschaftlicher Arbeiten (weitere Seminararbeiten oder Diplomarbeit, aber auch sonstige fachlich-technische Berichte) erwerben, aber auch durch positive Kritik solcher Arbeiten (Führungs- und Teamfähigkeit) zum Gesamtgelingen beizutragen.

Methodisch wird dieses Seminar in Form eines durch die Seminarteilnehmer getragenen Workshops durchgeführt, in dem überprüft werden soll, welche Bedeutung „Klassiker“ der Informatikliteratur aus heutiger Sicht haben. Während der „Entwicklungsphase“ wird die Kommunikation weitestgehend über MOODLE durchgeführt.

Das Workshop selbst ist zweigeteilt. Zu Semestermitte sollen in einem Kurzvortrag die Konzepte des gewählten Basisartikels kurz vorgestellt werden und ein Ausblick auf die eigenen Überlegungen dazu geboten werden. Im Hauptvortrag im Juni sind diese eigenen Überlegungen sodann ausführlich darzulegen.

Seminarthemen

„Klassiker“ gibt es in nahezu jedem Teilgebiet der Informatik. Um eine gewisse Vorstrukturierung zu bieten, sind die auf der Liste der „TRACES-Artikel“ genannten Beiträge in die Gruppen

- p_ Programming and Programming Languages
- s_ Software Specification
- t_ Software Technology
- h_ Software Development Process (hat etwas mit h-umans zu tun)
- d_ Data Bases and Information Systems
- k_ Knowledge Representation
- o_ Operating Systems

gegliedert.

Entsprechend Ihrem Interesse wählen Sie sich aus einer dieser Gruppe in Thema (manchmal ist dieses durch zwei zusammengehörige Artikel, gekennzeichnet durch gleiche Nummer mit Suffix a oder b definiert).
Sie können sich beliebig viel Arbeiten ansehen, bevor Sie eine wählen. Die getroffene Wahl teilen Sie den anderen Seminarteilnehmerinnen und Teilnehmern durch Posten einer entsprechenden Nachricht ***„x#, <Autor(en)>, <Titel>“*** mit. Durch Absenden dieser Nachricht haben Sie diesen Artikel **für sich reserviert** – und sich im Gegenzug verpflichtet, zu diesem Artikel eine Ausarbeitung zu verfassen.

Überlegen Sie sich also im Vorfeld, welchen Artikel Sie wählen wollen und prüfen Sie vor der eigenen Reservierung im Forum, ob dieser Artikel noch frei ist. – Zögern Sie aber auch

nicht zu lange mit Ihrer Entscheidung. Sie können ab sofort wählen. Am 12. März 07 wird diese Möglichkeit um 12:00, Mittag, geschlossen, sodass ab diesem Zeitpunkt die Themenvielfalt für das Seminar fest steht.

Terminplan

Siehe „TRACES-Basisinformation“

Beurteilungsschema

Beurteilt wird die auf Individualleistung und Mitarbeit beruhende Gesamtleistung bestehend aus

- Qualität deskommentierten Inhaltsverzeichnis 20 %
- Leistungen im Reviewprozess
  . Begutachtung d. kommentierten Inhaltsverzeichnis und
  . Begutachtung d. Endfassung 20 – 25 %
- Endfassung
  . Erstversion und Endfassung 40 %
- Kurzvortrag und Vortrag
  20 – 25 %

Der Ursprung Strukturierter Programmierung

Martin Glawischnig
0160357
m2glawis@edu.uni-klu.ac.at

Abstract

Strukturierte Programmierung – ein Konzept, das dem prozeduralen Paradigma der Programmierung entstammt – begleitet uns seit den 1960er Jahren. Der Grundgedanke besteht darin, für einzelne Teile des Programms lediglich die grundlegenden Strukturen Sequenz, Wiederholung und Entscheidung in hierarchischer Art und Weise zu verwenden.

Dieser Artikel soll nun einige Fragen rund um diesen Bereich beantworten: Wie ist das Konzept der strukturierten Programmierung eigentlich entstanden? Wird es heute immer noch angewendet? Welche Relevanz hat es im Vergleich zu damals?

1. Einleitung

Zu Beginn dieser Arbeit steht eine grundsätzliche Erklärung, was unter strukturierter Programmierung eigentlich verstanden werden kann. Im zweiten Kapitel wird erarbeitet, welche Vor- und Nachteile eine Anwendung dieses Konzepts mit sich bringt. Der letzte Teil beschäftigt sich mit der Entwicklung von strukturierter Programmierung bis heute. Dabei werden auch einige für dieses Thema relevante Programmiersprachen berücksichtigt.

1.1. Was ist eigentlich strukturierte Programmierung?

Im Allgemeinen kann gesagt werden, dass es sich bei strukturierter Programmierung um ein Konzept der Softwareentwicklung handelt. Der Begriff selbst, der E. W. Dijkstra zugeschrieben werden kann, ist heutzutage weit verbreitet. Doch woher stammt er, und was bedeutet er eigentlich? [11]

Als Urväter des Konzepts an sich sind die Mathematiker C. Böhm und G. Jacopini zu sehen, welche 1966 das Schreiben von Programmen auf das Verwenden von drei grundlegenden Kontrollstrukturen reduzieren konnten. Die Strukturen selbst und ihre Bedeutung werden in einem späteren Kapitel näher erläutert. Böhm und Jacopini hatten ihre auf Flussdiagrammen basierende Arbeit bereits selbst ausreichend belegt, die grundsätzliche Methodik wurde jedoch 1972 von H. Mills nochmals aufgegriffen und bewiesen. Es kann auch gezeigt werden, dass jedes bestehende Programm in eine derartige Darstellung überführt werden kann. Das Ergebnis mag nicht dieselbe Effizienz oder eine ähnlich gute Lesbarkeit besitzen, unterscheidet sich aber nicht von der Funktionalität an sich. [03, 14]

Dijkstra schlug 1968 unter der Bezeichnung „*structured Programming*“ ein Konzept zur Entwicklung von Programmen vor, welches diverse zu der Zeit aufkommende Probleme lösen sollte. Er selbst hat jedoch nie genau eingeschränkt, was der Begriff genau bedeutet, oder gar eine klare Definition abgegeben. Dijkstra meint, strukturierte Programmierung sei als Konzept zu sehen, das den Programmierer dabei unterstützt, mit jeglicher Komplexität bei der Entwicklung von Programmen umzugehen [09].

D. Bates hat dieses Konzept treffend in Worte gefasst:

„*What was originally meant by the term structured programming was the philosophy of structuring programs in such a way as to make them more intellectually manageable and more amenable to a convincing proof of correctness.*“ [12]

1.2. Motivation

Um diese Entwicklung nachvollziehen zu können, muss man zuerst verstehen, mit welchen Problemen damalige Entwickler zu kämpfen hatten. N. Wirth hat 1974 einen der wichtigsten Punkte treffend zusammengefasst:

„*A systematic, orderly, and transparent approach is mandatory in any sizable project nowadays, not only to make it work properly, but also to keep the programming cost within reasonable bounds.*“ [01]

Die Kosten für Software waren zu jener Zeit um einiges geringer als diejenigen für die dazugehörige

Hardware. Deshalb versuchte man auch, die kostspielige Rechenzeit optimal auszunutzen. Jedes Programm und jeder Algorithmus wurden bis ins kleinste Detail beleuchtet und optimiert. Dadurch entwickelten sich mit der Zeit clevere Tricks, welche für andere oft kaum durchschaubare Code-Konstrukte zur Folge hatten. Durch die rasante, vor allem aber auch preisliche Entwicklung der physischen Bauteile gerieten die Programmierer bald in ein Dilemma: Die Rechner der neuen Generation waren viel leistungsfähiger, aber auch um einiges komplizierter zu programmieren. Die resultierenden Programme wuchsen zu regelrechten Softwarepaketen heran, weil gleichzeitig immer mehr Funktionalität bereitgestellt werden konnte und sollte.

Programmierung drehte sich nicht mehr darum, kleine Stücke von Software möglichst effizient niederzuschreiben. Sie drehte sich darum, immer komplexere Systeme in immer kürzerer Zeit zu produzieren. Die Herausforderung und Schwierigkeit bestand nun eher darin, die Komplexität der entstehenden Systeme zu verarbeiten und die Vielzahl an unterschiedlichen Tätigkeiten zu verwalten.

Deshalb wurde es nun notwendig, Software so zu produzieren, dass sie von Anfang an möglichst fehlerfrei ist. Dies ist nachweislich um einiges kosteneffizienter, als im Nachhinein Unmengen von Bugs auszumerzen.

Die so entstandene Krise in der Softwareindustrie schrie förmlich nach einem Verfahren, mittels dem möglichst effizient viel korrekter Code produziert werden könnte. [06].

N. Wirth bezeichnet Effizienz sogar als wertlos, wenn man sich auf die Programme und den enthaltenen Code nicht verlassen könne. Für ihn stehen *„reliability“* von Software und die Fähigkeit, große und komplexe Programme organisieren zu können, im Zentrum von strukturierter Programmierung. [01]

2. Strukturiertes Programmieren

2.1. Grundlagen

Es gibt viele unterschiedliche Sichtweisen von strukturierter Pogrammeriung, aber auch einige Grundprinzipien, die von mehreren Vertretern dieses Konzepts gemeinsam vorgeschlagen werden, und auf welche im Folgenden genauer eingegangen wird:

2.2.1. Zielsetzung. Das grundsätzliche Ziel von strukturierter Programmierung ist das Erarbeiten von korrektem Programmcode mittels einiger mehr oder weniger einfacher Regeln. Es handelt sich nicht um eine definierte Vorgangsweise, und es gibt auch kein „Rezept“, nach dem man vorzugehen hat. Bei strukturierter Programmierung handelt es sich um ein Konzept, das – wenn man sich daran hält – bestimmte das Erlangen bestimmter Vorzüge und die Vermeidung bestimmter Nachteile verspricht. Diese werden im nächsten Kapitel näher erläutert. [01, 06, 14]

2.2.2. Vorgangsweise. Als grundlegende Vorgangsweise wird die Top-Down Entwicklung von Programmcode herangezogen. Das bedeutet, dass man von einer prägnanten, aber korrekten Beschreibung der Funktionalität ausgehend diese abstrakte Beschreibung iterativ erweitert. Die Erweiterungen beschreiben eine Eigenschaft, einen Teil der Funktion oder eine Designentscheidung auf der aktuellen Stufe. Der Vorgang wird immer genauer so lange wiederholt, bis die Blöcke zum Schluss durch echte Codezeilen ersetzt werden können. Wenn bei diesen Verfeinerungen davon ausgegangen werden kann, dass das ursprüngliche, abstraktere Konstrukt konzeptuell korrekt war, und wenn keine zusätzlichen Fehler eingeschleust wurden, so ist das Ergebnis ebenfalls korrekt. Wichtig dabei ist, dass man bei der Umsetzung in jedem Schritt dafür sorgt, dass das entstandene Programm genau das tut, was es soll – nicht weniger, aber auch nicht mehr! Jeder Schritt hat im eigentlichen Sinn eine Beschreibung des Programms zur Folge, mit fortschreitenden Iterationen wird diese Beschreibung eben immer genauer. [11, 09]

2.2.3. Kontrollstrukturen. Bei dem Prozess der Verfeinerung dürfen lediglich drei verschiedene „Building Blocks“ eingesetzt werden, die – von der verwendeten Programmiersprache abhängig – jedoch in verschiedenen Varianten existieren können.

– Die **Sequenz** (Konkatenation) ist eine Menge von Anweisungen, die in einer bestimmten Reihenfolge hintereinander ausgeführt wird.
– Die **Wiederholung** (Schleife) beschreibt das Verfahren, dass ein bestimmter Block (also auch zum Beispiel eine Sequenz) unter bestimmten Bedingungen wiederholt wird. Die Überprüfung dieser Bedingungen kann zu Beginn („while-do-Schleife“) oder am Ende („repeat-until-Schleife“) der Wiederholung stehen. Befindet sich die Überprüfung am Ende, so wird der umschlossene Block zumindest ein mal durchlaufen. Auch eine zuvor definierte Anzahl an Wiederholungen bietet eine Variante für eine Schleife („for-Schleife“).
– Beim dritten Baustein, der **Entscheidung**, wird die Ausführung eines Blocks von einer bestimmten Bedingung abhängig gemacht („if-then“ oder „if-then-else“).

Alle dieser drei Kontrollstrukturen haben wesentliche Eigenschaften gemeinsam: Sie haben genau einen Eintrittspunkt sowie genau einen Austrittspunkt, was das Beweisen der Funktionalität erheblich erleichtert.

2.2.4. Modularität. Ein wichtiger Aspekt von strukturierter Programmierung ist eine vernünftige Granularität der einzelnen Programmteile. Logisch zusammengehörige Blöcke sollten in eigenen Funktionen oder Modulen abgelegt werden, was bei entsprechendem Design eine spätere Wiederverwendbarkeit (Reuse) erheblich erleichtert. Hier ist jedoch Vorsicht geboten: Zu lange Passagen (so genannter „Spaghetticode“) werden schnell unüberschaubar, ebenso kann man bei zu vielen zu kurzen Funktionen schnell den Überblick verlieren.

Auch auf konzeptueller Ebene ist darauf zu achten, dass die Module untereinander eine vernünftige Beziehung zueinander erhalten. Die 1968 von Constantine eingeführten Begriffe *„coupling and cohesion“* (Kopplung und Kohäsion) sollten dabei ebenso im Vordergrund stehen wie das Prinzip des *„Information Hiding“*. Diese wesentlichen Konzepte hier näher zu erläutern würde jedoch den Rahmen dieser Arbeit sprengen. [09]

Laut Mills gehört zu diesem Thema jedoch auch die gute Lesbarkeit von Programmcode. Diese kann unter anderem durch eine gute und vor allem einheitliche Einrückung der einzelnen Programmzeilen erreicht werden. Einzelne zusammengehörige Blöcke sollen so auf derselben textuellen Ebene stehen, also gleich viele Leerzeichen weit eingerückt sein. Dies sollte sich durch die zu verwendende Top-Down Entwicklung fast schon von alleine ergeben oder zumindest erheblich erleichtert werden. Für funktionale Blocks wird auch die textuelle Länge einer Seite vorgeschlagen, da hier alle strukturellen Abhängigkeiten auf einen Blick erkannt werden können. [14]

2.2. Auswirkungen strukturierter Programmierung

Durch die Anwendung dieses Konzepts hat man mit einigen Folgen zu rechnen, deren Vorzüge und Nachteile im nächsten Abschnitt genauer erläutert werden.

Die Top-Down Entwicklung erfordert bzw. ermöglicht, dass zu jedem Zeitpunkt das zu entwickelnde Programm korrekt ist. Eine verständliche, natürliche Darstellung kann und sollte so lange wie möglich beibehalten werden.

Die einzelnen Schritte der Verfeinerung erfordern jeweils mehr oder weniger explizite Designentscheidungen. Diese sind jedoch später leicht nachzuvollziehen, da alle „Zwischenstationen“ leicht dokumentierbar werden. Jeder einzelne Schritt sollte im Idealfall eine korrekte Darstellung des Programms darstellen, was bestimmte Überprüfungen und auch Beweise zu fast jedem Zeitpunkt der Entwicklung zulässt.

Durch das Verfahren können die einzelnen Ersetzungen auch für Modularisierung herangezogen werden – jeder Schritt ermöglicht es, den eingesetzten Block durch einen Funktionsaufruf oder gar ein ganzes Modul zu ersetzen. Diese Module können für eine spätere Wiederverwendung in eigenen Bibliotheken abgelegt werden, um dieselbe Funktionalität an mehreren Stellen einzusetzen. Des Weiteren können die einzelnen Module durch neuere bzw. effizientere Varianten ausgetauscht werden, ohne negative Nebeneffekte zu erzeugen.

Trotz der Strukturiertheit, Top-Down Entwicklung und Modularisierung bedeutet das nicht, dass ein resultierendes Produkt qualitativ hochwertig einzustufen ist. Ein Programm kann nur dann als gut angesehen werden, wenn es nicht voll von Fehlern ist! Im Nachhinein das Programm durchzutesten und zu versuchen, alle Fehler auszumerzen ist jedoch nicht immer zielführend. Dazu meint Dijkstra:

„... program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.“ [06]

Aus dieser Überlegung heraus wird auch der Vorschlag aufgeworfen, dass man Software von Anfang an fehlerfrei entwickeln solle – dies braucht zwar einen aufwändigeren Prozess, jedoch würde dies insgesamt einiges an Test und Debugging ersparen. Strukturierte Programmierung beschreibt genau einen derartigen Prozess. Hier ist es möglich, ein Programm und dessen funktionellen Nachweis quasi parallel zu entwickeln. Dies eliminiert jedoch nicht die Notwendigkeit des Testens eines fertigen Produkts.

Ein weiteres der am meisten sichtbaren Ergebnisse von strukturierter Programmierung ist, dass das Ergebnis (also der entwickelte Code) völlig frei von goto Statements ist, die nicht zur standardmäßigen Abarbeitung der vorgeschriebenen Kontrollstrukturen benötigt werden.

2.3. Was hat das goto Statement damit zu tun?

Man muss sich vor Augen halten, zu welcher Zeit und in welcher Umgebung von Programmiersprachen das Konzept der strukturierten Programmierung gerade aufgekommen ist. Der führende und gebräuchlichste Vertreter war wohl immer noch Assembler, jedoch rückten 3GL-Sprachen wie Cobol und Fortran immer

mehr in den Mittelpunkt. Zu der Zeit lud jedoch noch kaum eine bestimmte Sprache zu einer strukturierten Vorgangsweise ein. Im Gegenteil, entsprechende sprachliche Konstrukte wurden zum Teil erst im Nachhinein oder in Folgeversionen in die einzelnen Sprachen integriert.

Die Verwendung von goto Statements (also direkten Sprüngen von einer Codezeile im Programm zu einer anderen) stand an der Tagesordnung. Zusätzliche Schleifendurchgänge oder überflüssige Vergleichsoperationen wurden möglichst eingespart. In den meisten Fällen führte dies zwar zu effizientem Code, die Lesbarkeit und auch die Wartbarkeit ließen jedoch zum Teil zu wünschen übrig.

Man hatte jedoch bereits erkannt, dass die Anzahl der im Quellcode verwendeten goto Statements indirekt proportional zu dessen Qualität zu sehen ist. Das bedeutet, dass je mehr goto Befehle ein Programm oder Codestück enthält, desto eher müssen Lesbarkeit, Wartbarkeit, Beweisbarkeit oder gar die Funktionalität darunter leiden. [01, 09]

Durch das Aufkommen von strukturierter Programmierung hat sich sehr schnell die Meinung verbreitet, es sei nichts anderes als goto-frei zu programmieren. Hier hat Mills prägnant reagiert:

„Structured programs should be characterized not simply by the absence of goto's but by the presence of structure.“ [14]

Die Eliminierung von goto Anweisungen aus bestehendem Code ist heute fast automatisch durch eine alternative Darstellung mittels Sequenz, Wiederholung und Entscheidung möglich. Es gibt heutzutage auch schon Softwarepakete, welche alte, teure Programme in eine neue Sprache übersetzen, um sie auf neuer Hardware wieder lauffähig zu machen. Bei einer einfachen Umsetzung ist jedoch nicht zu erwarten, dass das Ergebnis leichter zu lesen oder effizienter in der Abarbeitung ist. Wirth belegt dies in [01] anhand einiger aussagekräftiger Beispiele.

Dies soll jedoch nicht bedeuten, dass strukturierte Programmierung ganz ohne Sprunganweisungen auskommen soll – im Gegenteil, diese Anweisungen geben einem Rechner die einzige Möglichkeit, komplexe Kontrollstrukturen abzuarbeiten. Die Existenz solcher Befehle soll hier ausschließlich vom Level der Abstraktion abhängen – je näher man der Maschinensprache kommt, desto eher findet man gotos. Auf der niedrigsten Ebene werden auch die Basisblocks durch bedingte und unbedingte Sprünge im Maschinencode dargestellt. Man sollte sie jedoch in höheren Programmiersprachen nicht direkt verwenden. [04]

An dieser Entwicklung kann man erkennen, dass sich die grundsätzliche Aufgabe, die ein Programmierer bei seiner Arbeit zu bewältigen hat, mit der Zeit grundlegend verändert hat. Früher wurden teure Großrechner mit Programmen gefüttert, welche möglichst ideal auf alle Gegebenheiten abgestimmt waren. Die Maschine wurde bis ins letzte ausgereizt. Bei jeder Variablen wurde um Bits gefeilscht und versucht, sie möglichst häufig im Programm irgendwo zu verwenden. Mittlerweile versucht man eher, eine Programmiersprache dazu zu nutzen, um komplexe Systeme zu generieren. Deshalb unterscheidet man heute auch zwischen *„coding“*, der reinen, systemnahen Produktion von Programmcode, und *„programming“*, dem systematischen Erarbeiten von Algorithmen. Dass dieser Unterschied auch für das Erlernen von (strukturierter) Programmierung eine zentrale Rolle spielt, wird in einem späteren Kapitel im Detail beleuchtet. [01]

3. Vorteile und Nachteile von strukturierter Programmierung

Im Folgenden möchte ich aufzeigen, was strukturierte Programmierung bringen kann, wenn diese Technik vernünftig eingesetzt wird. Dazu werden die einzelnen Vorteile und auch die Nachteile, die aus einer Anwendung erwachsen können, gegenübergestellt. Es ist jedoch rasch zu bemerken, dass hier die Vorteile wohl überhand nehmen.

3.1. Vorteile

3.1.1. Top-Down Entwicklung. Durch diese Art der Softwareentwicklung können diverse Vorzüge erreicht werden. Zum Ersten bietet sich die Möglichkeit, die Wartbarkeit des Endproduktes zu erhöhen. Einerseits sind alle eingearbeiteten Designentscheidungen automatisch dokumentiert, und andererseits können Nebeneffekte bei Änderungen bereits im Vorfeld erkannt oder grundsätzlich vermieden werden. Zum Zweiten wird, wie bereits weiter oben näher erläutert, der entstandene Code automatisch modularisiert, was neben einer besseren Lesbarkeit und Wartbarkeit auch die erhöhte Möglichkeit zur Wiederverwendung mit sich bringt.

Beide Punkte bieten einem Team von Entwicklern vor allem eines, nämlich eine erhebliche Zeitersparnis: Sei es durch die bereits modularisierten Bibliotheken, oder durch die parallel erarbeitete Beweisführung für die Funktion, oder auch durch den wesentlich verringerten Aufwand der Fehlerbehebung.

3.1.2. Weniger Fehler. Durch die angewendete Methode der iterativen Verfeinerung werden von vornherein weniger Fehler produziert und in das System einge-

schleust. Dies ist dadurch erklärbar, dass man sich bei den einzelnen Schritten auf eine beschränkte Aufgabe konzentrieren kann und nicht das Konzept des ganzen Systems im Hinterkopf behalten muss. Wenn man von einer fehlerfreien Ausgangsposition (zum Beispiel einer anfänglich fehlerfreien Spezifikation der Anforderungen) ausgeht, wird zumindest konzeptuell ein fehlerfreies Ergebnis erzeugt. Dies gilt jedoch nur dann, wenn in den weiteren Schritten keine Fehler eingebracht werden!

Auch bei nachträglichen Änderungen können mögliche Fehler durch unerwünschte Seiteneffekte vermieden oder wenigstens frühzeitig erkannt werden.

Weniger Fehler steigern die Qualität des Produkts und das bedeutet auch, weniger Zeit für Debugging aufwenden zu müssen. Und das wiederum senkt den Gesamtaufwand und die Kosten.

3.1.3. Handhabbarkeit. Beim Vorgang der Abstraktion eines komplexen Problems sorgt ein strukturiertes Vorgehen dafür, dass der menschliche Intellekt nicht mit zu komplexen Konstrukten überfordert wird.

„Building Blocks“ für strukturierte Programmierung haben alle genau einen Einstiegspunkt und einen Austrittspunkt. An diesen können die einzelnen Blocks miteinander verbunden werden. Diese Punkte lassen sich jedoch auch als Zustände in einem Zustandsdiagramm beschreiben, an denen Variablen Werte aus bestimmten Wertebereichen enthalten sollen oder sogar müssen. Derartige „Assertions“ sind für den Nachweis der Funktionalität eines Codestücks sowohl hilfreich als auch notwendig. Fortgeschrittene Programmiersprachen unterstützen dieses Konzept sogar als integrierte sprachliche Konstrukte.

Des Weiteren können die eindeutigen Zugangspunkte auch zur Unterstützung der Stabilität herangezogen werden. Durch die Top-Down Entwicklung haben Variablen automatisch nur denjenigen Scope, den sie brauchen, und keinen größeren – die Verwendung ein und derselben Variablen an verschiedenen, unabhängigen Codestücken wird unterbunden. Dieses Ergebnis erhält man dadurch, dass Variablen erst dann eingeführt werden (dürfen), wenn sie auch wirklich benötigt werden. Globale „Hilfsvariablen“ schon zu Beginn zu definieren, mit dem Hintergedanken, sie dann irgendwann zu verwenden, wäre eben nicht strukturiert. Dies ist vor Allem auch dann hilfreich, wenn Speicherplatz in dauerhaft laufenden Programmen freigegeben werden muss, um die Stabilität des Systems zu gewährleisten. [01]

3.1.4. Leichtere Beweisführung. Dadurch, dass der gesamte Entwicklungsprozess vorgegeben ist, kann – wenn man sich von Anfang bis zum Ende rigoros daran gehalten hat – die Funktionalität des entstandenen Programms leicht bewiesen werden. Dies unterstützt letztendlich die Zuverlässigkeit des Systems. Der Großteil der Beweisführung wird sogar quasi als Nebenprodukt des eigentlichen Prozesses mitgeliefert. Wesentlich dafür ist jedoch, dass die ursprüngliche abstrakte Beschreibung, die den Ausgangspunkt darstellt, genau das beschreibt, was als Ergebnis erwartet wird. Dasselbe gilt auch für die nachfolgend getroffenen Designentscheidungen. [05]

3.1.5. Modularisierung. Das Top-Down Verfahren lässt ein Softwareprodukt auch sehr leicht modularisiert entstehen. Das bedeutet, dass ein in einem Schritt der Verfeinerung entstandener Block leicht in eine eigene Subroutine beziehungsweise ganze Bibliotheken ausgelagert werden kann. Die Eigenschaft der eindeutigen Ein- und Ausstiegspunkte erleichtert dies vor allem auch bei der Wahl der notwendigen Parameter. Dies unterstützt Lesbarkeit, Wartbarkeit und natürlich auch die Wiederverwendung der einzelnen Module. Überflüssige Parameter werden ebenso vermieden, wie Module, die mehr Funktionalität enthalten, als sie eigentlich sollten, und so für eine spätere Verwendung eher unbrauchbar werden.

3.1.6. Compiler. Ein interessanter Punkt ist auch die Tatsache, dass einige der Nachteile, die im Folgenden beschrieben werden, zumindest teilweise durch einen vernünftigen Einsatz von Compilern wieder wett gemacht werden können: Tricks, die dazu verwendet wurden, zum Beispiel mittels komplexer goto Konstrukte eine enorme Steigerung der Effizienz zu erreichen, können durch einen Compiler (oder auch durch einen Precompiler) automatisch eingebaut werden. Übersetzter (zumindest diejenigen einiger der moderneren Programmiersprachen) verwenden bereits automatische Codeoptimierer. Der Vorteil davon ist, dass die Entwicklung und der endgültige Code dennoch strukturiert und lesbar bleiben.

3.2. Nachteile

3.2.1. Voraussetzungen. Damit strukturierte Programmierung den gewünschten Erfolg zeigen kann, müssen jedoch bestimmte Punkte eingehalten werden. Das wichtigste ist wohl, dass jede Abstraktionsebene so gehalten ist, dass das beschriebene Programm genau das tut, was es soll – nicht mehr und nicht weniger! Darauf ist bei der ersten Ebene, von der das Verfahren startet, besonders zu achten. Ist die anfängliche Beschreibung ungenau oder gar unzutreffend, wird das

erarbeitete Ergebnis nicht die gewünschten Resultate erzielen.

3.2.2. Entwicklungszeit. Die schrittweise Verfeinerung mag – vor allem, wenn jede Ebene ausreichend dokumentiert wird – aufwändiger erscheinen, als den Code direkt niederzuschreiben. Dies ist vor allem dann der Fall, wenn man sich für eine schöne, aber nicht strukturierte Lösung eine Alternative einfallen lassen muss. Wirth beschreibt so in [01] ein passendes Beispiel („*Selecting Distinct Numbers*“), bei dem die erste, natürliche Formulierung des gefragten Algorithmus eine nahezu selbstverständliche goto Anweisung enthält. Daraufhin wird jedoch eine intuitiv ebenso verständliche Lösung desselben Problems erarbeitet.

Dieses Manko wird jedoch durch die oben angeführten Vorzüge wieder wettgemacht.

4. Die Entwicklung

4.1. Das Software Paradoxon

Software ist ein Produkt. Es wird designed, von Programmierern produziert und anschließend gewartet. Doch im Vergleich zu anderen Produkten sind hier Qualität und Preis nicht mehr direkt proportional zueinander! Software, die nicht von Anfang an einer vernünftigen Architektur und einem durchdachten Entwicklungsprozess unterliegen, sowie Systeme, bei denen sich laufend Fehler einschleichen, haben eine viel höhere Entwicklungszeit und einen erhöhten Wartungsaufwand, um etwaige Bugs zu beseitigen. Dies alles treibt die Kosten immens in die Höhe.

Wie im bisherigen Verlauf dieser Arbeit gezeigt wurde, kann die Entwicklung gut strukturierter Programme dem entgegenwirken. Die schrittweise Verfeinerung erlaubt, Entscheidungen erst dann zu treffen, wenn sie wirklich notwendig sind. Dies können Design-Entscheidungen sein, oder auch Formen der Darstellung für die verwendeten Datentypen. Was dies für die verwendete Programmiersprache bzw. das eingesetzte Paradigma (zum Beispiel prozedural oder objektorientiert) bedeutet, wird in einem späteren Abschnitt genauer beleuchtet.

4.2. Programmiersprachen

Das Aufkommen und die Entwicklung strukturierter Programmierung hatten einen immensen Einfluss auf die gängigen Programmiersprachen. Ein Grund dafür war wohl auch die Tatsache, dass es zur Erstellung strukturierter Programme irrelevant ist, welche Programmiersprache verwendet wird – manche Sprachen unterstützen dies eben mehr, manche weniger. [11]

Bei bereits bestehenden Sprachen (wie zum Beispiel FORTRAN oder ALGOL) konnte eine zwingende „Unterstützung“ wohl kaum nachträglich eingebaut werden. Doch mit der Zeit wurde versucht, mehr und mehr strukturierte Programme zu entwickeln. Nachfolger bzw. Erweiterungen ermöglichten jedoch eine leichtere Umsetzung des Konzepts. IFTRAN zum Beispiel wurde als Vorstufe zu FORTRAN eingesetzt, um die Verwendung mächtigerer Kontrollstrukturen zu ermöglichen. [02]

Neu entstandene Sprachen, wie zum Beispiel PASCAL, waren bereits so konzipiert, dass die Programmierung von gut strukturiertem Code unterstützt wurde. PASCAL unterstützte jedoch immer noch den goto-Befehl an sich. Dies ermöglicht dem ambitionierten Programmierer, Strukturen, die nicht direkt in der Sprache abgebildet sind, selbst nachzuahmen. Beispiele dafür wären der Ausstieg aus der Ausführung einer Schleife oder die Nachbildung eines Konzepts für ein Exception-Handling. Dies kann zwar für die ganze ALGOL-Familie gesagt werden, aber auch PASCAL hat sich hier nach dem bestehenden Trend gerichtet.

Neben den verwendeten Programmiersprachen haben aber auch zusätzliche Werkzeuge, welche für diese Sprachen entwickelt wurden, einiges zum Thema beigetragen. Tools, die heutzutage zur Entwicklung von Software eingesetzt werden (wie etwa die Eclipse Platform) unterstützen die Ausarbeitung gut strukturierter Programme immens. Diverse Hilfsmittel helfen dabei, den Code strukturiert aufzubauen und zu halten. Ein Beispiel dafür ist die Automatische Quellcodeeinrückung, die vor allem für eine gute Lesbarkeit sorgt. So werden alle Zeilen entsprechend ihrer Zugehörigkeit zu übergeordneten Blocks eingerückt und zu lange Zeilen an geeigneten Stellen umgebrochen. [13]

4.3. Wie hat sich goto gehalten?

Moderne Sprachen, wie zum Beispiel JAVA, bieten bereits ausreichend viele Befehle, um alle Möglichkeiten für einen Kontrollfluss abzubilden. Neben der Möglichkeit, aus Schleifen auszusteigen, gilt wohl das Exception-Handling als Paradebeispiel. Es gibt jedoch auch Kritiker, die meinen, dass diese Konstrukte nichts anderes sind als eine andere Bezeichnung für goto. In JAVA wurde sogar „goto“ als Schlüsselwort ohne Funktion reserviert.

Nachdem auf der Ebene des Maschinencodes Sprungbefehle prinzipiell wesentlicher Bestandteil für zwei der grundlegenden Bausteine (nämlich Wiederholung und Entscheidung) darstellen, kann man derartige

Konstrukte dann auch in nahezu jeder Programmiersprache finden. Lediglich die Vielfalt und die Benennung unterscheiden sich zum Teil gravierend.

Dijkstra offerierte deshalb einen Vorschlag zur Klassifizierung von Konstrukten, die seiner Meinung nach erlaubt sein sollten: Er beschreibt einen vom Programmierer unveränderlichen Zeiger auf den sich dynamisch verändernden Programmfortschritt. So gibt es zu jedem Zeitpunkt eine Menge von Koordinaten, die eindeutig die Aufrufkette bis zur aktuellen Codezeile nachvollziehen lässt. Laut Dijkstra sollen nur solche Befehle erlaubt sein, die eine derartige Zuordnung zulassen – wie auch immer sie benannt sind. [05]

4.4. Ausbildung

J. F. Schrage hat schon 1980 in [08] beschrieben, dass Studenten mit Grundlagen in strukturierter Programmierung bei Wartungsarbeiten wesentlich besser arbeiten, als welche ohne diesen Hintergrund.

Dies bestätigt, wie wichtig es ist, beginnenden Programmierern die Bedeutung dieser Thematik näher zu bringen – ihnen zu vermitteln, welche Vorzüge gut strukturierte Programme mit sich bringen. Beginnen muss dies damit, die Auszubildenden „*programming*" und nicht „*coding*" zu lehren. [01]

Hier ist jedoch wesentlich, vor allem Neulingen die Bedeutung strukturierter und die Probleme nicht strukturierter Programmierung aufzuzeigen, um ein grundlegendes Verständnis dafür aufzubauen. Des Weiteren ist es auch sinnvoll, goto-Statements (bzw. dessen Abwandlungen) nicht einfach zu verbieten, sondern beizubringen, wie und wann man welche Konstrukte sinnvoll einsetzen kann und darf. [08]

4.5. Beispiel: SFC Editor

Im Folgenden möchte ich ein Beispiel für ein Werkzeug darstellen, welches die Relevanz der Thematik bis in die heutige Zeit belegt. Der aus dem Jahr 2004 stammende Artikel [10] von T. Watts beschreibt einen grafischen Editor, welcher aus den zu Beginn beschriebenen „*Building Blocks*" komplexe Flussdiagramme erzeugen lässt. Die so dargestellten Programme sind nach dem hier dargestellten Konzept vollständig strukturiert, da der Editor keine wie auch immer gearteten Ausnahmen zulässt – es können lediglich bestehende Blocks erweitert werden.

Neben dem grafischen Teil wird automatisch auch eine textuelle Darstellung generiert, die je nach Einstellung entweder aus Pseudo-Code oder sogar aus kompilierbarem C++-Code besteht. [10]

In Abbildung 1 ist ein Bildschirmfoto des Editors zu sehen, welches die grundsätzliche Funktionalität anzeigt. Auf der linken Seite findet sich ein Diagramm, welches ausschließlich an den einzelnen Kombinationspunkten (durch dargestellt kleine graue Kreise) erweitert werden kann. Dadurch wird immer ein strukturiertes Programm erzeugt. Auf der rechten Seite findet sich dann die Pseudo-Code Darstellung des Algorithmus.

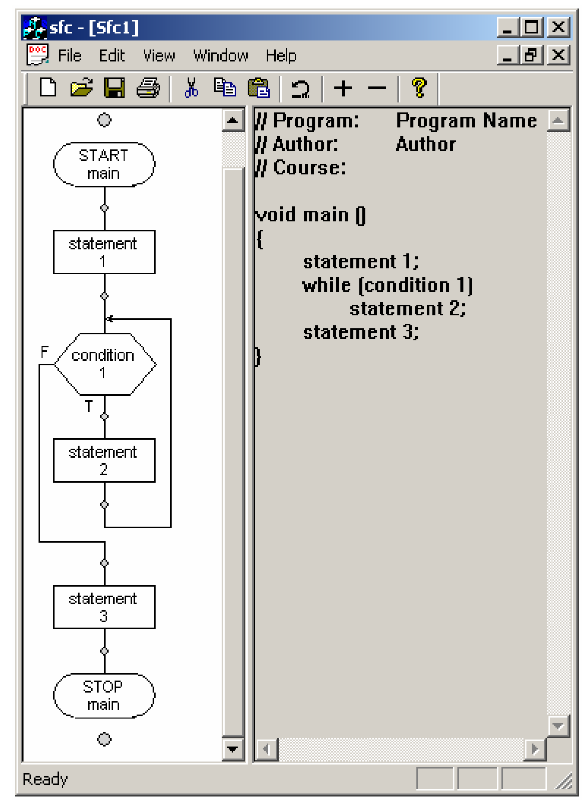


Abbildung 1. Bildschirmfoto des SFC Editors

5. Resumee

Zu Beginn dieser Arbeit stand eine Einführung, was unter strukturierter Programmierung überhaupt zu verstehen ist, und welche Grundlagen ihre Basis bilden. Von einer genaueren Beschreibung, was strukturierte Programme ausmacht spannt sich der Bogen über den Einfluss des goto-Statements bis hin zu den Vor- und Nachteilen dieser Vorgehensweise.

An der aufgezeigten Weiterentwicklung und dem Zusammenhang mit den Programmiersprachen wurde die bis heute zunehmende Relevanz der Thematik dargestellt.

Meiner Meinung nach ist es jedoch wichtig, dass neben dem hier beschriebenen „*programming*" in wei-

terer Zukunft das „*coding*" ebenso eine zentrale Rolle spielen wird. Neben der Aufgabe, auch weiterhin Compiler und andere maschinennahe Systeme zu schreiben, wird es – heute mehr noch als früher – immer mehr von den Ressourcen her eingeschränkte Geräte geben, die einer adäquaten Programmierung bedürfen. Kühlschränke, die selbst online nachbestellen können und intelligente Kleidungsstücke sind hier wohl erst der Anfang. Derartige Geräte und Tools werden jedoch optimierte Programme benötigen, um die eingeschränkte Hardware optimal zu nutzen. Hierbei können dann wohl auch die „alten" Tricks dabei helfen, die gewünschte Effizienz zu erreichen, obwohl hier sicher eine Überführung eines zuerst strukturiert entwickelten Programms eine angebrachte Strategie ist.

Nichtsdestotrotz kann ich mich nur der Meinung von D. McCracken aus [07] anschließen, dass strukturierte Programmierung neben der Einführung von „*closed subroutines*" eine der größten Erfindungen der Softwaretechnologie ist und bleibt.

6. Referenzen

Basisartikel

[01] Wirth, N. 1974. On the Composition of Well-Structured Programs. *ACM Comput. Surv.* 6, 4 (Dec. 1974), 247-259.

weitere Referenzen

[02] Bezanson, W. R. 1975. Teaching structured programming in FORTRAN with IFTRAN. In *Proceedings of the Fifth SIGCSE Technical Symposium on Computer Science Education* D. T. Bonnette, Ed. SIGCSE '75. ACM Press, New York, NY, 196-199.

[03] Böhm, C. and Jacopini, G. 1979. Flow diagrams, Turing machines and languages with only two formation rules. In *Classics in Software Engineering*, E. N. Yourdon, Ed. ACM Classic Books Series. Yourdon Press, Upper Saddle River, NJ, 11-25.

[04] Creak, A. 2003. Everything is Fortran, in its own way. *SIGPLAN Not.* 38, 4 (Apr. 2003), 7-12.

[05] Dijkstra, E. 1979. Go to statement considered harmful. In *Classics in Software Engineering*, E. N. Yourdon, Ed. ACM Classic Books Series. Yourdon Press, Upper Saddle River, NJ, 27-33.

[06] Dijkstra, E. 1979. The humble programmer. In *Classics in Software Engineering*, E. N. Yourdon, Ed. ACM Classic Books Series. Yourdon Press, Upper Saddle River, NJ, 111-125.

[07] McCracken, D. 1979. Revolution in programming: an overview. In *Classics in Software Engineering*, E. N. Yourdon, Ed. ACM Classic Books Series. Yourdon Press, Upper Saddle River, NJ, 173-177.

[08] Schrage, J. F. 1980. Educator's view of structured concepts. In *Proceedings of the ACM 1980 Annual Conference* ACM '80. ACM Press, New York, NY, 327-341.

[09] Tomayko, J. E. 1990. Anecdotes. *IEEE Ann. Hist. Comput.* 12, 4 (Oct. 1990), 269-276.

[10] Watts, T. 2004. The SFC editor a graphical tool for algorithm development. *J. Comput. Small Coll.* 20, 2 (Dec. 2004), 73-85.

[11] Weiner, L. H. 1978. The roots of structured programming. In *Papers of the SIGCSE/CSA Technical Symposium on Computer Science Education* (Detroit, Michigan, February 23 - 24, 1978). ACM Press, New York, NY, 243-254.

Sekundärliteratur

[12] Bates, D. 1976. Structured Programming, *Infotech State of the Art Report*, Infotech International Limited, Maidenhead, England.

[13] Mills, H. D., Top Down Programming in Large Systems. In *Debugging Techniques in Large Systems*, B. Rustin, Ed., Prentice-Hall, 1971 in [11]

[14] Mills, H. D., Mathematical Foundations for Structured Programming In *IBM Report* FSC 72-6013, 1972 in [11]

Internetquellen

http://en.wikipedia.org/wiki/Structured_programming, zuletzt besucht: 18. Mai 2007

7. Acknowledgment

Abschließend möchte ich allen Reviewern danken, die durch ihre Hinweise sehr zur Qualität dieser Arbeit beigetragen haben.

Are goto's don't do's?

Ingomar Preiml
0260592
ipreiml@edu.uni-klu.ac.at

Abstract

The discussion about the use of goto's in programming languages is quite old. It had its peak with Dijkstra's well known arguments against goto statements [1]. At a later point Knuth [2] discussed this issue with different points of view in order to ease the discussion. A question that is going to be answered is how the goto discussion had an influence on the use of goto's and on other programming constructs over time. Additionally a few contemporary ways of how goto statements are used and should be used to develop well structured programs will be looked at. In this context it will become evident that the use of goto statements can have a positive effect on the program structure when used in limited ways.

1. Introduction

First of all I would like to start with the definition of the term "goto statement". A goto statement is a branching construct to transfer the control flow of a program to a specific label in an unconditional manner. It depends on the programming language how such a label may look like (e.g. a name followed by a colon or simply a line number). Goto is a rather old programming construct and is a basic building block in assembler languages. For being international I would like to mention that goto is a common Japanese surname as well [26].

Back in the days goto's have been widely programming construct. As the problems that had to be solved became more complicated, the resulting programs became more complex too. Dijkstra [1] mentioned that goto's will have a devastating effect on the structure of a program, if used inappropriately. Because of this statement a lot of rumor arose about using goto statements and many algorithms with the purpose to replace goto's with other constructs (e.g. while, case, ...) were devised [15][16][17][18][19]. In 1974 Knuth and many others decided to analyze the use of goto statements again and came up with the idea to design programs in two steps. In the first step the program is written without goto's and with structure in mind, even if this would be inefficient. Afterwards the well documented and well understood code should be automatically transformed into a more efficient variation, using goto's if appropriate. The resulting code might not be too readable but that won't matter because the original source will be used to refine or explain the program. This shall be the starting point of my analysis of how the use of goto statements developed over time and it has to be said that this discussion very controversial.

A long time has passed since the arguments about goto statements were laid to rest but it definitely had an impact on the use of goto statements and the design of programming languages. One effect was the omitting of goto's in several programming languages, e.g. Modula-2, Oberon, Java, Python and others. As a note on the side, Java 1.5 does not provide the functionality of a goto command but the keyword "goto" is reserved. The above mentioned languages are all known to be part of the imperative programming language paradigm. Another paradigm of programming languages known as functional or logical languages do lack goto's overall. Yet another effect of the goto discussion was the development of several unconditional branching constructs with limited scopes like *break, continue* and *return*. In contrast there are still a few programming languages that provide goto's, e.g. C, C#, C++ and Visual FoxPro. Further down I am going to analyze two widely used open source programs in terms of their utilization of goto's.

The last chapter will describe a few software engineering aspects and suggestions of how goto statements and similar unconditional branching constructs might be used productively.

2. What goto's were

In former times goto's were a common tool for programming code and were an important part of the solution of many algorithmic problems. But with the onset of increasingly powerful computers new and more powerful algorithms were devised. With more power came more complexity [14], leading to more advanced programming languages and to more

complex source code as well. Foreseeing several problems with the program structure if goto's are used wrongly, Dijkstra [1] made a comment leading to a grim discussion about the usability of goto's in common.

2.1. Dijkstra's perspective

Dijkstra pointed out the negative effects of goto statements, that can easily disrupt the structure of a program and therefore opted for the abolishment of goto's from higher level programming languages. It has to be said that this argument was issued in a time where it was most common to use goto's in many programing languages (e.g. COBOL, ALGOL, C, ...).

To begin with, Dijkstra outlined two facts. Firstly, the programmer is only capable of defining a program that is then executed dynamically. The dynamic process has to conform with the programs requirements. Secondly, humans are more accustomed to understanding static contexts rather then dynamic process situations. Therefore it is beneficial to correlate the program structure (the code) and the dynamical process (the program being executed) as much as possible. This correlation can be expressed in using coordinates to describe the current state of the running program. This aids in the determination of the value of a variable at any given program state. Thus making it clearer how the running program will behave when looking at the source code.

There are several programming building blocks that have to be considered when setting up a system of coordinates to describe a running program. First, there is the simple sequence of commands. Here, it is enough to have a program counter to get hold on to the actual program state. Another building block are decision taking constructs (e.g. *if*, *case*, ...) which also just require a program counter to be described. In contrary it is more complex to describe a program state when procedures are included. Say, there needs to be a counter for the position in the caller procedure and an additional counter for capturing the position inside the called procedure. This may result in an array of counters dependent on the procedure calling stack. The very moment we introduce repetition constructs (e.g. for, while, ...) we have to introduce additional counters to keep track of the count of repetitions. With the above mentioned structural parts we are able to uniquely determine a program state using a set of program and loop counters thus providing a good way to analyze several program conditions (e.g. when does a variable hold a certain value).

Given the fact that we introduce the goto statement, we are able to disrupt the flow of control which in the absence of goto's is much more regulated and well formed. With goto's it can be hard to find an unique set of coordinates to describe the actual progress of the program. Of course it is possible to use the program counter to obtain an unique program state, but this state is much more meaningless in terms of understanding the program.

Dijkstra is not altogether against the use of goto's when they do not harm the program's structure but he was convinced that it is easier to abuse goto's than to use them in a sensitive way.

2.2. Knuth's perspective

Knuth [2] had a different point of view in his article about structured programming with goto's. He proposed two contrary ideas. First he introduced several techniques to write iterations and error exits without the need of goto's. Second he suggested to write well structured but maybe inefficient programs without goto's and transfer these into more efficient ones by replacing certain programming constructs with goto's and labels. The resulting code is certainly less readable but the original program is well documented and serves as a basis for further development.

Knuth confessed that he was biased because wrote a lot of code examples in his books include goto's. Nevertheless he tried to analyze a number of problems deciding whether they can be written without goto's efficiently or not. Among others loops, error exits, hash coding and text scanning without goto's seemed to be possible without any loss of performance. Other constructs proofed to be more readable and efficient with goto's (e.g. recursion elimination, multi way branching, removal of boolean variables, tuning considerations).

To start with loops seemed to be somehow reasonable. It is easy to understand that critical program loops should be written as efficiently as possible. Without using goto's it is necessary to design loops in a way, that as much operations as possible are moved outside of the loop (e.g. boundary checks, ...). Error exits (e.g. if x>max goto overflow) might be handled by a smart compiler. If not, validity checks should be moved outside of the loops if possible. Hash codes should be extremely effective because they can be issued very often and are therefore highly bound to a low execution time. It has been proven that it is better to design a hash function in a well structured way and tune it with local goto's to obtain maximum performance, than to write it from scratch with goto's.

A few other problems are also performance critical and should not be done without goto's like text searching. Recursion elimination is another main feature of performance tuning. The overhead of handling a caller stack for all the called procedures can

be significantly high. Indeed it is not arguable that recursions can provide very small and nice solutions for a number of problems but the negative side effect of being slower in most cases cannot be neglected. Another application for goto's is the removal of boolean variables using code duplication and unconditional jumps. The code depending on a boolean decision is copied and altered in that way, that the first copy handles the true-case and the second part handles the false-case.

After all, the real issue is to write programs that are well structured and can be proven (formal or informal), i.e. it is enough to really understand whether a program is correct or not. For this a good structure is supportive.

To conclude his essay, Knuth stated that it is not a good idea to write programs with goto's and then eliminate them. The goal should be to write understandable programs, not only in a small but in a bigger scale. A good way to achieve this is by using abstraction in different levels of the program, to provide a means of understanding the whole program conveniently. This conclusion does not exclude the use of goto's by default. As long as the goto statements do not destroy the possibility to easily grasp the big picture. Moreover if goto's provide an even better and more understandable code everything should be fine. The mentioned modularity and abstraction can nowadays be found “per default” in object oriented languages that are more focused on the issue of program structure. After all Knuth did choose a more pragmatic way to analyze the goto issues than Dijkstra and his findings were aimed to rather smoothen the discussion by providing both points and contra points for and against the use of goto's. His proposal for transforming well structured code into efficient code using goto's can be seen as the counterpart to the goto elimination having its roots in the same epoch and arose out of the same discussion.

2.3. Goto elimination

Although the process of goto elimination is very old it is still needed for re-engineering legacy software [15][19]. The removal of goto's is a precondition for porting legacy code to a new programing language that does not have control structures directly equal to goto's (e.g. Java).

2.3.1. Program equivalence. Due to the fact that a code transformation alters a program code but should not alter its functionality, there needs to be a definition of equivalence to ensure that the altered program behaves like the original. In fact there are a few kinds of equivalences defined. The input-output equivalence [16] is the weakest form because it allows the program structure to be altered significantly. The only restriction is that the output of both programs must be the same. Another type is flow-graph equivalence [17] which does not include that much freedom as the input-output equivalence. Two algorithms are flow-graph equivalent if their program flow is equivalent, leaving room for reordering and additional commands. The strongest form of equivalence is the structural equivalence defined by Ramshaw [18]. Two programs are considered structural equivalent if they are flow-graph equivalent and one program can be transformed into the other simply by replacing a few components of the source code and without altering the codes ordering. There are a few preconditions to be met in order to be able to achieve certain kinds of equivalences (e.g. no head-to-head crossings, ...). Those preconditions can be achieved using other code transformations [15].

2.3.2. Elimination algorithms. There exist several algorithms for replacing goto's with other programming constructs, mainly conditions and loops. I am not going to describe the goto elimination algorithms themselves but I will outline the key informations necessary to make the algorithms understandable. Please have a look at the quoted papers for additional information and examples.

The algorithms proposed in [16] utilizes flowcharts to describe the program structure, a very common way of designing programs back in the days. The algorithm works on the assumption that every goto statement can be replaced by a suitable while loop. In more detail, the flowchart is broken up into blocks with only one exit. The aim for every block is to only have a single top level decision which can easily be transformed into a while statement. In case this is not possible, additional boolean variables have to be inserted to keep track of the program flow. The disadvantage of this algorithm is that it provides an input-output equivalence only, altering the structure of the original program significantly.

Another way of getting rid of goto's is described in [15]. In this thesis the program code is represented in a special program algebra (PGA) for an easier reasoning about the correctness of certain concepts. Furthermore there are two techniques mentioned to remove goto's with the use of additional variables and without. The more interesting algorithm is definitely the removal without requiring additional variables thus resulting in a transformation that is almost structural equivalent. To achieve this several steps are necessary. Firstly, so called head to head crossings need to be removed. A head to head crossing are overlapping goto statements that have to be removed by reordering the code. Another step of the algorithm is the replacement of

goto's with loops with multiple exits. In this step the whole program is put into a loop and whenever the program exits, the loop is terminated. There are a few more steps included in this algorithm, but they are mainly refinement steps that need not to be explained here.

The last algorithm I would like to explain can be found in [19]. This paper resulted of a software to remove goto statements from source code with several thousand lines written in COBOL. It is based on finite automata and regular expressions. The algorithm starts with writing down the program as flowchart. The flowchart representation can be transferred easily into a finite state automaton, by replacing the edges of the flowchart with states and the symbols of the flowchart with transition conditions. The next step is to transform the automaton into regular expressions. This is done by writing down every path from the program start to the program termination as a regular expression. All paths are concatenated to the regular expression representation of the program. A few reductions on the regular expression lead to a minimal representation of the code which can then be transformed into the final program using loops instead of goto statements.

2.3.3. Final comments on goto eliminations. Goto eliminations are somehow complex program transformations. But they are nevertheless an integral part of reverse engineering efforts in connection with legacy programs. The above mentioned algorithms are based on theoretical work which is several decades old but is still present in research and practical work.

A current example of an implementation is the Reasoning Systems' Refine tool [19] written in LISP and designed for removing goto's of COBOL source code. Another example is a patch-set for the McCAT C compiler which enables the removal of goto's via a simple command switch [4].

As we can see, the discussion whether goto's are a good way of doing things or maybe dangerous for a program's structure, have had an impact on the way things are done a long time ago but are still a matter of present research. Therefore, I would like to outline a few more impacts of goto's in the next chapter of my paper.

3. Goto's in present times

In today's education system goto's are affected with a sense of dirtiness. More particularly the whole goto discussion lead to the establishment of a taboo [13]. A decade after the Dijkstra's letter [1], the use of goto's was evermore unwanted and the designer of programing languages backed away of including unconditional jumps and labels in their new languages. Nevertheless the act of goto avoidance was the corner stone of the development of today's more sophisticated programming structures.

A negative effect of the abolishment of goto's is that the technology of writing code with explicit flow of control is deemed to die. This leads also to the problem, that low level programming (assembler, hex coding, ...) is no longer understood nor taught anymore. As a result it is harder for today's programmer to write efficient programs because the low level knowledge of the program flow near the hardware gets lost. In contrary this argument is moderated by the fact that modern compilers became better in optimizing code than those of former times. Still there are a number of programs utilizing goto's in a well structured and fashionable manner with a deeper focus on structure than on optimization.

3.1. The appearance of goto's in today's programs

I am going to analyze two quite big open source programs written in the programming language C. The programs I had in mind are the GNU/Linux kernel (version 2.6.21) and the GNU C compiler collection (version 4.2.0). This analysis is not meant to be a performance measurement. The main goal of this observation is to outline patterns how goto's might be used in today's programs without the fear of destroying a good structure.

3.1.1. The GNU/Linux kernel. The first program I am going to look at is the kernel of the Linux operating system [20], written in C and known for its high complexity. There have been a few discussions whether goto's are a feasible way of doing things [21], especially in the kernel of an operating system.

One perspective is that goto's are considered harmful by default, which might be a reason of the constant misuse in former times. The other opinion is that goto's, if used cautiously, are a good means of structuring the code in a nice and readable way. Most of the kernel programmers have agreed on that and the use of goto's is desired if they are applied in a reasonable way [22]. But let's have a look at a few samples of code from the latest kernel release.

The examined kernel was downloaded from [20]. There seemed to be a few situations where goto's are used in a considerable amount. First they are most often used for performing *cleanups*. The example I would like to present is a form of a cleanup-stack. This programming technique is used, for example, for initializing hardware drivers where several conditions might fail and a previous state has to be restored,

before returning to the calling procedure again (see illustration 1).

Exception handling is another use for goto's because the programming language C does not provide a built-in support for handling errors. As described in illustration 2, there might be several points in the program, where errors can occur. The only thing that needs to be done is to branch to the exception handler that is usually located at the end of the procedure after the default exit.

```
int error = -EINVAL;
/* Monitor the error conditions and go to the cleanup
 * routines if necessary */
if (!strlen(class_dev->class_id)) goto out1;

error = kobject_add(&class_dev->kobj);
if (error) goto out2;

error = make_deprecated_class_device_links(class_dev);
if (error) goto out3;
    ...
/* If there was no error go to directly to return */
goto out1:
/* Undo several settings in a cleanup stack */
out3:
    class_device_remove_groups(class_dev);
out2:
    if(parent_class_dev) class_device_put(parent_class_dev);
    class_put(parent_class);
out1:
    class_device_put(class_dev);
    return error;
```

Illustration 1: Example of a cleanup stack with goto's.

Another usage for goto's is the *termination of multiple nested loops*, where a single break command is not sufficient. The main idea is to exit both loops without adding more complexity to the termination rules of the loops.

```
/* Check several conditions and in case of an
 * exception call the exception handler */
if (class == ATA_DEV_ATA) {
    if (!ata_id_is_ata(id) && !ata_id_is_cfa(id))
        goto err_out;
} else {
    if (ata_id_is_ata(id))
        goto err_out;
}
...
/* Default exit, no error occured */
*p_class = class;
return 0;
/* Exception handler */
err_out:
    if (ata_msg_warn(ap))
        ata_dev_printk(dev, KERN_WARNING, "failed to IDENTIFY "
        "(%s, err_mask=0x%x)\n", reason, err_mask);
    return rc;
```

Illustration 2: Example of an exception handler.

There are a few other usages of goto's in the Linux kernel like rather nasty backward jumps that are discouraged as we will see later. Overall it can be said, that the emphasis of the usage of goto's is rather concentrated on issues of the program's structure than it is on performance. This conclusion is assisted by the fact that the amount of goto's in the Linux kernel increased steadily with the increasing of the size (have a look at diagram 1).

3.1.2. The GNU C compiler collection. The second free available and very big software project I am going to look at is the GNU C compiler collection (gcc). The compiler is available at [24] and I examined the newest compiler with the version 4.2.0. First of all I would like to mention, that the use of goto's in the GNU C compiler is similar to the patterns already described in the GNU/Linux section.

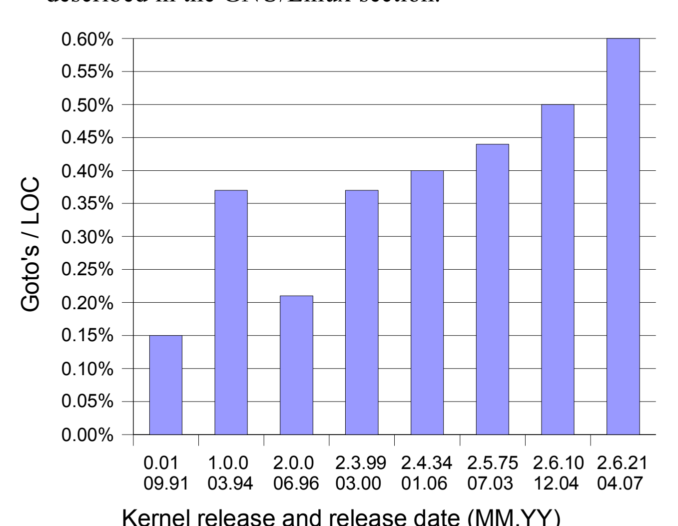


Diagram 1: Goto's in the GNU/Linux kernel.

There are also cleanup processes, exceptions as well as inner loop exits present in the source code. But there are also other usages for goto's like loop constructs. The only example I would like to give here is a part of the gcc scanner which utilizes goto's as a recovery mechanism (illustration3).

```
/* initial label to return back to in case
 * a new line is read */
retry:
c = ' ';
c = skip_spaces (fp, c);
if (c == '\n') {
    source_lineno++;
    lineno++;
    goto retry;
}
...
/* read an integer token and exit */
if (ISDIGIT (c)) {
    ...
    c = INT_TOKEN;
    goto done;
}
...
/* exit label */
done:
...
return c;
```

Illustration 3: A part of the gcc scanner.

This code construct is very likely to be generated by a compiler compiler and is then filled with meaningful code.

The examination of the amount of goto's used in the gcc (see diagram 2) showed that there are more than twice as much goto's in the GNU/Linux kernel than there are in the gcc source code. Additionally the amount of goto's is not increasing but is rather stable.

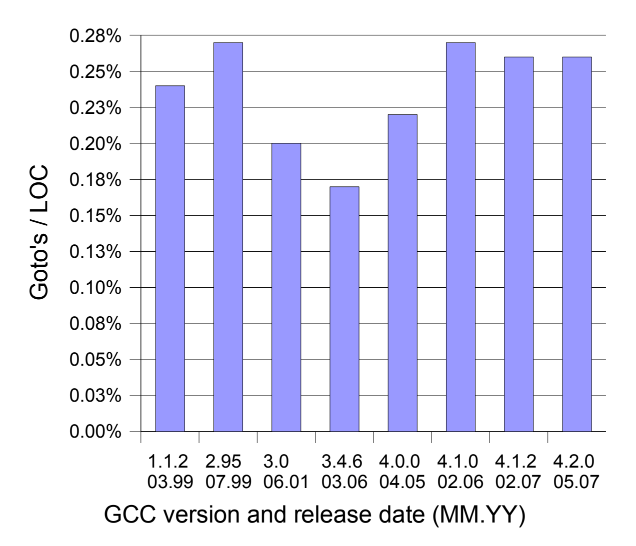


Diagram 2: Usage of goto's in the GCC compiler.

To conclude this subpart it has to be said that although goto's are not really present in “higher” programming languages like Java, but they still have a field of application, at least in programming with C. Additionally, it is important to know how to use the program construct goto in a sophisticated way to not destroy or obfuscate the program structure.

Another interesting fact is, that goto-like branching constructs have survived in almost every programming language except functional and logical languages, but they have different names and are limited to a certain kind of jumps. Those constructs are going to be explained in the next chapter.

3.2. The goto discussion and “modern” programming languages

Returning from the depths of using goto's as programming constructs, other effects causing the initial goto discussion a few decades ago will be looked at. One result was the development of several algorithms to automatically remove goto's from source code (see section 2.3). Another result was the replacement of goto branches with other similar named and similar working constructs like “come froms” [24] or “leave <label> with <expression>” [25].

In the long term several other explicit branching constructs with a narrower scope, like *break*, *continue*, *case* and *return* were developed and proved to be good-natured to the programing structure. Yet another impact of the goto discussion was the development of imperative programming languages without any goto's or goto like branching constructs (Modula-2, Oberon, Java, Python, ...) [5][6][7][8]. It has to be mentioned that there are also languages supporting unconditional branching (C, C++, C#, Visual FoxPro, ...) [9][10][11][12]. The question arises whether so called goto-less programming languages are really lacking means for unconditional branches. The answer is a clear and simple no, as I am going to describe in the next sub-chapter.

3.2.1. A programming algebra with labels and goto's. It is quite easy, to show that certain kinds of loop termination constructs, case constructs and even return constructs are nothing more than limited unconditional branches. To achieve this, I would like to use a few components of a simple program algebra with labels and goto's (PGLE) [15]. To start with, a few PGLE building blocks are described in table 1.

Building block	Representation	Explanation
Basic instruction	Every a ∈ Σ*	Denotes a simple instruction
Termination instruction	!	Terminates the program
Positive test instruction	+a for each a ∈ Σ*	If instruction a is evaluated with true, the subsequent instruction after a is executed
Negative test instruction	-a for each a ∈ Σ*	If instruction a is evaluated with false, the subsequent instruction after a is executed
Forward jump instruction	#k whereby k ∈ N	A jump over k operations
Concatenation	;	For modeling an instruction sequence
Label instruction	£k where k ∈ N	A label with an assigned natural number
Goto instruction	##£k where k ∈ N	Jumps to the label denoted by £k

Table 1: A short description of a PGLE.

3.2.2. Concerning goto like constructs. The constructs *break*, *continue* and *case* are now going to be examined. The proofs are based on a simple transfer of a pseudo-code like programming language into PGLE notion. There is one thing to mention beforehand: it is simpler to transform do-while loops into PGLE because the loop condition needs not to be altered. For simplicity only do-while loops are used in the following examples.

Table 2 shows the translation of a do-while loop with an if-break statement included. In case the if statement is validated with true, the loop is going to be terminated. The translation shows, that the code can be described with two simple goto's, the *break* statement is one of them. It has to be said, that *break* is more constrained than goto, because it can only jump to the end of a loop and nowhere else.

Pseudo code	PGLE
do statement1; if condition1 break; while condition2 exit;	£0; statement1; +condition1; ##£1; +condition2; ##£0; £1; !;

Table 2: A *break* construct in PGLE.

The *continue* command is almost identical to a *break* command with the only difference, that the program flow branches to the start of the actual loop (table 3).

Pseudo code	PGLE
do statement1; if condition1 continue; statement2; while condition2 exit;	£0; statement1; +condition1; ##£0; statement2; +condition2; ##£0; !;

Table 3: A *continue* command in PGLE.

A different command is the *case* construct which can be translated into a multi-level branching structure and can then be reduced to a goto-based form (table 4).

Pseudo code	PGLE
select var; case a: statement1; break; case b: statement2; break; end select; exit;	 +var=a; statement1; ##£0; +var=b; statement2; ##£0; £0; !;

Table 4: A select-case structure in PGLE.

It is also not too difficult to derive a PGLE representation of a try and catch block. This might be done with a label followed by the catch block and a condition for every line in the try block, to check whether an exception has occurred or not.

It is evident that there exists a simple transformation of loop constructs to constructs goto with goto's and labels. The above shown transformations serve the purpose to increase the awareness that a lot of widely used programming languages are nothing more than goto's limited to a certain scope. The limited scope is well known and can be handled easily to achieve a good program structure. Unconditional program transfers (other than goto statements) are very limited in present programming languages because program language designers decided so to overcome the above mentioned problems with program structure.

Nevertheless goto's are present in a number of languages and there are a few software engineering aspects that should be considered here. Additionally, “constrained” goto's (*break*, *continue*, *case*, ...) are subject to these software engineering aspects as well.

4. Software Engineering aspects

It is very important to understand that goto statements can be a means of creating several important features like exception handling in low level programming languages like C. It is also obvious that a misuse of unconditional branching will lead to programs with rather bad structure. Therefore a few pragmatic rules [3] will be introduced for determining how goto statements can be used in a good way. Moreover, those rules can be applied to modern unconditional branching constructs as well because of their clear relationship to goto statements.

First, no goto should ever jump into a loop or into a conditional statement. A jump into a loop or a condition will lead to head-to-head crossings, producing interwoven program blocks without a clear structure. Second, the label a goto is going to jump to has to be located in the same sub-program, i.e. in the same procedure or function (do not use long-jumps to exit procedures). The third and last rule demands that every jump has to be forward directed, that means the corresponding goto's have to appear before the labels that they jump to. This definitely excludes *loop constructs with goto's* as well as the *continue* command for controlling loops.

The three simple rules provide a way to use goto statements in a good way. They ensure a program to be split up in building blocks that can be understood and handled easily thus providing a good program structure.

5. Conclusion

The discussion about use and abuse of the goto command is very old and had a lot of impacts on how programs are written nowadays. A very big research area was coupled to the goto elimination which still is interesting for reverse engineering of legacy applications. Nevertheless goto's are present in a number of today's programs like the GNU/Linux kernel and are accepted as a programming construct. Another impact of the afore mentioned discussion was

the development of languages without explicit goto statements but with a few other unconditional branching constructs (*break*, *continue*, *case*, ...). To use goto statements in a sensitive way it is necessary to limit unconditional branching to a few cases that have been described as software engineering aspects. Finally, goto's are still present in today's programming languages in one form or another and they clearly influenced the knowledge we have today.

6. References

[1] E. W. Dijkstra (1968). *Goto's considered harmful.* Communications of the ACM, Vol. 11, No. 3, pp. 147-148.

[2] D. E. Knuth (1974). *Structured programming with goto statements.* ACM Computing Surveys (CSUR), Vol. 6, No. 4, pp. 261-301.

[3] W. Gellerich, Kosiol M. and Ploederer E. (1996). *Where does GOTO go to?* Proceedings of the 1996 Ada-Europe International Conference on Reliable Software Technologies, pp. 385-395.

[4] A. M. Erosa (1994). *A goto-elimination method and its implementation for the McCAT C compiler.* Master thesis, McGill University. Retrieved on 10.05.2007 from: http://citeseer.ist.psu.edu/erosa95gotoelimination.html

[5] *Java reference manual*. Retrieved on 11.05.2007 from: http://java.sun.com/reference/index.html

[6] *Python reference* manual. Retrieved on 11.05.2007 from: http://docs.python.org/ref/ref.html

[7] *Oberon reference.* Retrieved on 11.05.2007 from: http://www.oberon.ethz.ch/compiler/

[8] *Modula-2 reference.* Retrieved on 11.05.2007 from: http://www.modula2.org/reference/index.php

[9] E. Huss (1997). *The C library reference* guide. Retrieved on 11.05.2007 from: http://www.acm.uiuc.edu/webmonkeys/book/c_guide/

[10] *C# specification*. Retrieved on 11.05.2007 from: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf

[11] *C++ reference*. Retrieved on 11.05.2007 from: http://www.cppreference.com/keywords/goto.html

[12] *Visual FoxPro reference*. Retrieved on 11.05.2007 from: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_foxhelp9/html/6dcddc3f-9944-4ad8-be2f-003610af616a.asp

[13] L. Marshall and J. Webber (2000). *Gotos Considered Harmful and Other Programmers' Taboos.* In A.F. Blackwell & E. Bilotta (Eds). Proc. PPIG 12, pp. 171-180.

[14] E. W. Dijkstra (1972). *The humble programmer.* Communications of the ACM, Vol. 15, No. 10: pp 859-866.

[15] T. D. Vu (2007). *Semantics and applications of process and program algebra*. Master Thesis, University of Amsterdam, Supervisor Prof. J.W. Zwemmer, pp 135-188.

[16] E. Ashcroft and Z. Manna (1972). *The translation of 'go to' to 'while' programs.* Information Processing, Vol. 71, North Holland, Amsterdam, pp. 250-255.

[17] W. W. Peterson, T. Kasami and N. Tokura (1973). *On the capabilities of while, repeat, and exit statements.* Communications of the ACM, Vol. 16, No. 8, pp. 503-512.

[18] L. Ramshaw (1988). *Eliminating go to's while preserving program structure.* Journal of the ACM Vol. 35, No. 4, pp. 893-920.

[19] P. Morris, R.A. Gray and R.E. Filman (1997). *Goto removal based on regular expressions.* Journal of Software Maintenance: Research and Practice, Vol. 9, No. 1, pp. 47-66.

[20] *The Linux kernel archives.* Retrieved on 12.05.2007 from: http://www.kernel.org/.

[21] *Linux: using goto in kernel source* (2003). Retrieved on 18.05.2007 from: http://kerneltrap.org/node/553/2131

[22] *"Goto im Kernel"* (2005). Retrieved on 18.05.2007 from http://www.lk.etc.tu-bs.de/lists/archiv/lug-bs/2005/04/msg00063.html

[23] *GCC, the GNU compiler collection*. Retrieved on 12.05.2007 from: http://gcc.gnu.org/

[24] L. Clark (1984). *A linguistic contribution to goto-less programming*. Communications of the ACM, Vol. 27, No. 4, pp. 349-359.

[25] W.A. Wulf (1972). *A case against the GOTO.* Proceedings of the ACM, Vol. 2, pp. 791-797.

[26] Common Surnames in Japan. Retrieved on 06.06.2007 from: http://www.rootsweb.com/~jpnwgw/Names.html

Model-Driven Software Development and Beyond

Eine geschichtliche Betrachtung des Artikels *Executable Object Modeling with Statecharts* von David Harel und Eran Gery

Seminararbeit für
(622.002) SB Seminar aus Angewandte Informatik, SS 2007
Alpen-Adria Universität Klagenfurt

Michael Steindorfer [0560854]

ABSTRACT

Bereits in den neunziger Jahren präsentierten David Harel und Eran Gery mit dem Artikel „Executable Object Modeling with Statecharts" einen Modellierungsansatz für reaktive Systeme, aufbauend auf Klassen- und Zustandsdiagrammen, welche in Verbindung die vollständige Synthese des Programmcodes erlaubten. Mit Rhapsody, einem Softwareprodukt der Firma i-Logix[1], stellten sie parallel dazu ein Werkzeug vor, welches dieses Vorgehensmodell verwirklichte. Es wurde in diesem Zusammenhang bereits der Gedanke vom „Model-Driven Software Development (MDSD)" umgesetzt, bevor dieser öffentlich formuliert wurde.

Diese Arbeit setzt die Konzepte aus dem Artikel von Harel und Gery in Bezug zur Gegenwart und hebt die daraus resultierenden Entwicklungen hervor. Es wird primär auf die Idee des „Model-Driven Software Development (MDSD)" und weiterführende Forschungsarbeiten der Autoren eingegangen, die dieses Konzept erweitern.

1. EINLEITUNG UND MOTIVATION

Welche Bedeutung hat der Artikel „Executable Object Modeling with Statecharts" [7] von David Harel und Eran Gery heute – zehn Jahre nach seiner Publikation – in der Scientific Community? Zu welchen Technologien, die für uns heute selbstverständlich sind, haben die Erkenntnisse der Autoren geführt bzw. beigetragen? Warum ist dieser Artikel noch heute von Bedeutung?

Diese Arbeit verfolgt das Ziel, die Konzepte von David Harel und Eran Gery anfangs offen zu legen und deren weitere Entwicklung in den letzten zehn Jahren zu betrachten. Dabei wird kein Anspruch auf Vollständigkeit erhoben. Vielmehr werden einzelne Konzepte der Autoren herausgegriffen und ihre Entwicklung im Laufe der Zeit weiter verfolgt.

In Kapitel 3 werden die Grundideen des „Model-Driven Software Development (MDSD)" im Bezug auf die Konzepte von David Harel und Eran Gery beschrieben. Dabei wird teilweise auf technische Aspekte eingegangen, mehr aber auf die Ideen an sich. MDSD ist ein sehr komplexes Gebiet und kann daher in dieser Arbeit nur ansatzweise bearbeitet werden. Es werden primär Konzepte behandelt, die in direkter Relation zum Artikel „Executable Object Modeling with Statecharts" [7] stehen. Eine gute Einführung in dieses Thema geben die Bücher „Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management" von Thomas Stahl und Markus Völter [11] und „Model Driven Architecture. Applying MDA to Enterprise Computing" von David S. Frankel [2]. [2]

Aufbauend auf den Darstellungen des Model-Driven Software Development wird ein Ausblick auf die weiterführende wissenschaftliche Bearbeitung diese Gebietes von David Harel gegeben, welche zum Teil weit über MDSD hinaus geht. Weiters bearbeitete Harel in seinen späteren Arbeiten auch offene Fragestellungen, die im Jahr 1997 noch nicht beantwortet waren.

2. EXECUTABLE OBJECT MODELING WITH STATECHARTS

Mit dem im Jahr 1987 publizierten Artikel „Statecharts: A visual formalism for complex systems" [4] behandelte David Harel erstmals Zustandsdiagramme (Statecharts) in der Form, wie sie auch heute noch existieren. Entworfen wurden sie mit dem Ziel, die Modellierung von komplexen, reaktiven Systemen zu erleichtern.

In der Mitte der 90er Jahre wurden mit dem Artikel „Executable Object Modeling with Statecharts" [7] erste Konzepte präsentiert, die heute unter dem Begriff „Model-Driven Software Development" zu finden sind. Zustandsdiagramme wurden in diesem Artikel aufgegriffen, um das dynamische Verhalten während der Laufzeit zu modellieren. Die Bestandteile des Entwicklungskonzeptes sind in Abbildung 2 grafisch veranschaulicht. Die grundlegende Idee ist, sich von der Programmierung zu lösen und weiter zu abstrahieren, um der steigenden Komplexität von Softwaresystemen Herr zu werden. Auf einer Metaebene wird zur Modellierung eine Kombination aus Zustands- und Klassendiagrammen verwendet, welche Informationen zum Laufzeitverhalten beinhaltet, die sonst nur im Programmcode zu finden sind. Dieses Konzept, welches die Verbindung von Verhaltenskomponenten und der Struktur beschreibt, wird als „ausführbares Modell" bezeichnet.

[1] Heute ist dies Firma unter dem Namen Telelogic bekannt.

[2] Das Buch „Model Driven Architecture. Applying MDA to Enterprise Computing" [2] von David S. Frankel behandelt den Standard der Object Management Group (OMG) und ist nicht als allgemeines MDSD-Buch zu betrachten.

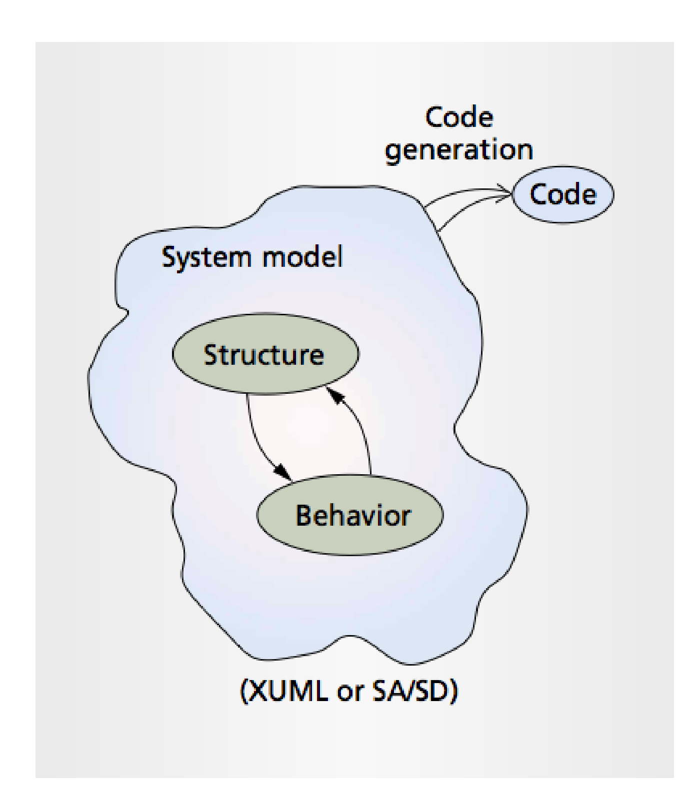


Figure 1: System-Modellierung mit Codegenerierung. Das Modell vom System beinhaltet Struktur- und Verhaltensinformationen, welche mittels visuellen Formalismen wie „executable UML“ (xUML) oder „structured analysis/structured design“ (SA/SD) dargestellt werden. Eine klar definierte semantische Basis ermöglicht die Generierung von vollständig lauffähigen Code aus dem Modell [6].

Die von David Harel und Eran Gery eingesetzten grafischen Notationen[3] und ihre Aufgaben werden im Folgenden kurz vorgestellt:

- Klassendiagramme spezifizieren die Struktur des Systems, in dem das System mittels Klassen, ihren Beziehungen untereinander und Rollen modelliert ist. Sie werden verwendet, um die Struktur – die Statik des Systems – zu modellieren.
- Zustandsdiagramme beschreiben das Systemverhalten zur Laufzeit. Es wird die Dynamik, das reaktive Verhalten, festgelegt. Ein Zustandsdiagramm verbunden einem Klassendiagramm spezifiziert alle Verhaltensaspekte von einem Objekt einer Klasse.
- Sequenzdiagramme – auch unter dem Begriff „Message Sequence Charts“ (MSCs) bekannt – wurden als optionaler Bestandteil gesehen und waren nicht Teil des Systems[4]. Die Autoren sahen damals MSCs als Mechanismus zur Reflexion, da diese gut geeignet sind um Szenarien und Zusammenhänge zwischen Objekten zu beschreiben.

[3] Als dieser Artikel verfasst wurde, war die Unified Modeling Language (UML) nicht sehr verbreitet, aber schon damals hielten die Entwickler ihre Notationen konsistent mit denen von UML.

Die Semantik und Notation der Diagramme muss exakt spezifiziert sein und darf keine Interpretationsmöglichkeiten offen lassen. So kann eine effiziente und automatisierte Synthese in Programmiersprache der dritten Generation wie z.B. C++ oder Java angewandt werden. Nennenswert und durchaus mit Potenzial verbunden ist die Tatsache, dass nicht nur Sourcecode sondern auch eine Hardwarebeschreibungssprache das Ergebnis einer solchen Transformation sein kann [6, 7].

2.1 Modellierung der Verhaltenssemantik mit Zustandsdiagrammen

Um das Verhalten einer Anwendung über die gesamte Laufzeit hinweg abzubilden, wurden in [7] zwei wichtige Punkte identifiziert: die Initialisierung am Beginn und die Dynamik während der Laufzeit. Diese Aufgaben werden folgend dargestellt:

- Bei der Initialisierung werden aus der Komponenten- und Klassenstruktur die von Beginn an existierenden Instanzen angelegt und in Verbindung gesetzt. Bei Beziehungen mit festen Kardinalitäten bzw. bei Kenntnis der Ober- und Untergrenzen, werden die Informationen zur Instanzierung aus dem Klassendiagram gewonnen. Bei nicht festen, numerischen Kardinalitäten muss diese Information auf eine andere Art abgebildet werden. Die Autoren entschieden sich für Initialisierungsskripten, welche einmalig am Beginn der Laufzeit ausgeführt werden und die Instanzierung der nötigen Objekte nach manuellem Editieren anstoßen.
- Zustandsdiagramme beschreiben wie Objekte miteinander kommunizieren und interagieren. Des Weiteren wird mit ihnen auch das objektinterne Verhalten dargestellt. Zur Modellierung der Dynamik während der Laufzeit werden bei den Zustandsdiagrammen Events und konventionelle Methodenaufrufe erlaubt. Mit Events lässt sich ein System auf einer hohen Abstraktionsebene gut modellieren, dafür ist die Übersetzung in Programmcode schwieriger. Methodenaufrufe hingegen erlauben eine sehr einfache Umsetzung, sind aber nicht so mächtig.

Vererbung und Verhaltensübereinstimmung [7]: Vererbung ist ein wichtiges und mächtiges Konstrukt im objektorientierten Paradigma, welches in Verbindung mit Zustandsdiagrammen einige Fragen aufwirft. Generell wird bei der Vererbung nur eine Übereinstimmung der Schnittstellen gefordert, die noch nichts über das Verhalten aussagen. Eine Klasse B, welche eine Subklasse von A ist, kann ihr Verhalten komplett ändern ohne gegen Vererbungsregeln zu verstoßen. Wenn aber – wie im Ansatz von David Harel und Eran Gery – ein Zustandsdiagramm zu der Klasse A zugeordnet ist, wie kann garantiert werden, dass dieses auch zu B passt?

[4] Sie werden hier nur der Vollständigkeit wegen aufgeführt, da ihnen bei weiterführenden Forschungsarbeiten von David Harel noch viel Bedeutung zugemessen wird.

Mit diesen Problemen, welche große Forschungsgebiete öffnen, sahen sich die Autoren damals konfrontiert. David Harel und Orna Kupferman widmeten sich Jahre später in [10] dieser Thematik, in welcher sie klare Definitionen aufführten, was genau unter Verhaltensübereinstimmung im Bezug auf Vererbung zu verstehen und wie komplex eine solche Prüfung sei.

2.2 Rhapsody - Heutige Positionierung des Entwicklungswerkzeuges

Die im Artikel [7] von David Harel und Eran Gery vorgestellte Entwicklungsumgebung, mit dem Namen Rhapsody, definiert sich heute als eines der führenden Werkzeuge für Model-Driven Software Develpment (MDSD)[5].

Die zu diesem Zeitpunkt aktuelle Version 7.1 unterstützt Code- und Modell-zentrierte Workflows, wobei mit Hand erstellter Code von Rhapsody automatisch in Modelle umgesetzt werden kann. Diese lassen sich im weiteren Verlauf zur Programmanalyse und als Dokumentation verwenden. Beim Modell-getriebenen Ansatz kann der Entwickler auf einer höheren Abstraktionsebene mit grafischen Modellen arbeiten und sich den Programmcode automatisiert generieren lassen.

3. MODEL-DRIVEN SOFTWARE DEVELOPMENT (MDSD)

Im Gegensatz zur Modell-basierten Entwicklung, bei der Modelle hauptsächlich das System dokumentieren, rücken diese beim „Model-Driven Software Development“ ins Zentrum. Dieser Ansatz kann als eine natürliche Fortsetzung der Programmierung, wie wir sie kennen, gesehen werden und abstrahiert diese Konzepte weiter[6]. Der Programmcode stellt eine andere Sicht auf das Modell[7] dar und soll so weit wie möglich automatisiert aus den Modellen synthetisiert werden [2, 11].

Thomas Stahl und Markus Völter beschreiben die Zielsetzung in [11] mit folgenden Worten: *„Das Ziel von MDSD ist es daher, domänenspezifische Abstraktionen zu finden und diese einer formalen Modellierung zugänglich zu machen. Dadurch entsteht ein hohes Automationspotenzial für die Fertigung von Software und dies wiederum lässt sich in den allermeisten Fällen in eine deutliche Produktivitätssteigerung umsetzen.“*

Model-Driven Software Development muss nicht als radikaler Paradigmenwechsel in der Softwareindustrie gesehen werden, sondern eher eine Konsolidierung einer Vielzahl von Trends und Technologien, welche die Softwareentwicklung in der letzten Zeit vorantrieben. Etablierte Konzepte wie zum Beispiel Objektorientierung, Component-based Development und Patterns werden durch MDSD nicht obsolet [2].

[5] Dieser Ansatz wird ausführlich in Kapitel 3 dargestellt.
[6] In [2] wird in der Einleitung, im Vorwort und im Chapter 1 diese Positionierung sehr gut dargestellt. Das Vorwort wurde von Michael Guttmann verfasst.
[7] Obwohl man in erster Linie an grafische Modelle denkt, sei erwähnt, dass diese nicht unbedingt grafisch sein müssen.

3.1 Abgrenzung zu Model-Driven Architecture (MDA)

Der Begriff „Model-Driven Architecture“ (MDA) ist in der Fachpresse sehr geläufig, wird aber sehr unterschiedlich verwendet. Nach Thomas Stahl und Markus Völter stellt MDA eine Standardisierungsinitiative der „Object Management Group“ (OMG)[8] zu Model-Driven Software Development dar, welche mit ihren Standards vorrangig die Ziele der Interoperabilität (Herstellerunabhängigkeit bei Entwicklungswerkzeugen) und Portablilität (Plattformunabhängigkeit) verfolgt. Durch diesen Fokus werden bei MDA im Gegensatz du MDSD auch viele Einschränkungen getroffen um diese Ziele zu erreichen. MDSD kann deshalb als eine Ausprägung oder Interpretation von Model-Driven Software Development gesehen werden [11].

3.2 Herausforderungen und Ziele

Bevor auf den Ansatz und einige technische Grundlagen eingegangen wird, werden die von MDSD verfolgten Ziele und die Herausforderungen im Bezug auf heute etablierte Paradigmen betrachtet, die anschaulich am Beginn von [11] dargestellt werden:

- Durch Automatisation wird eine Steigerung der Entwicklungsgeschwindigkeit angepeilt. Aus formalen Modellen wird durch Transformationsschritte in letzter Instanz lauffähiger Code erzeugt. Durch den Einsatz von Automatisation und formal definierten Modellen wird implizit eine Steigerung der Software-Qualität erreicht, da sich die Softwarearchitekuren adäquat in der Implementierung wieder finden.
- MDSD hilft auch beim Versuch, eine bessere Handhabbarkeit von Komplexität durch Abstraktion zu erreichen. Modellierung wird in den Rang der „Programmierung“ – auf eine höhere abstrakte Ebene – gehoben. Um dies zu erreichen werden problemorientierte Modellierungssprachen benötigt.
- Zwei weitere, sehr wichtige, Ziele, die von der „Object Management Group“ (OMG) verfolgt werden, sind die Interoperabilität zwischen Entwicklungswerkzeugen von verschiedenen Herstellern und die Portabiltät, welche ein Plattformunabhängigkeit mit sich bringt.

3.3 Der Ansatz

Modellierung ist kein neuer Ansatz und wird vor allem in den schwergewichtigen Entwicklungsprozessen verwendet um die innere Struktur zu dokumentieren. Konsistenzproblemen, welche zwansgsläufig eintreten, wird in den Prozessmodellen mit Reviews, etcetera entgegen gewirkt [11].

Beim Roundtrip- bzw. Reverseengineering, welches bereits in etliche Lifecycle-Entwicklungsumgebungen Einzug gefunden hat, entstehen diese Konsistenzprobleme nicht. Hierbei handelt es sich aber lediglich um Visualisierung des Sourcecodes (mittels UML z.B.). Daraus ergibt sich der Umstand,

[8] Die OMG ist ein weltweit offenens Konsortium, welches aus mehr als 800 Mitgliedern besteht. Bekannst ist sie z.B. für UML (Unified Modeling Language), CORBA (Common Object Request Broker Language) und IDL, MOF (Meta Object Facility) und XMI.

dass beide „Sichten“ sich auf dem selben Abstraktionsniveau befinden (im starken Gegensatz zu MDSD) [11].

Auf dem Niveau von Programmiersprachen sind Architekturmerkmale und Konstruktionsparadigmen oft nur noch schwer als solche zu erkennen, da das Abstraktionsniveau ein viel niedrigeres ist. Die Struktur liegt dadurch teilweise in verschleierter Form vor und ist nur mit Hilfe der Dokumentation nachvollziehbar [11].

In der modellgetriebenen Softwareentwicklung wird durch abstrakte und zugleich formale Modelle ein deutlich effektiverer Ansatz präsentiert. Mit abstrakten Modellen, welche sich durch Kompaktheit und Reduktion auf das Wesentliche beschränken, wird das nötige Abstraktionsniveau erreicht[9]. Durch die exakte formale Spezifikation erhalten die Modelle eine gleichwertige, semantische Aussagekraft wie der Programmcode. Dies ist so zu verstehen, dass aus den Diagrammen ein Großteil der Implementierung generiert werden kann [11].

Modelle werden somit zum Bestandteil des Systems selbst und nicht nur – wie in der modellbasierten Entwicklung – als Dokumentation angesehen. Diesem Umstand wird auch mit der Abgrenzung im Namen Rechnung getragen, in dem explizit auf „Modell-getrieben“ Bezug genommen wird. Durch dieses Potenzial wird die Softwareentwicklung erheblich beschleunigt und qualitativ erheblich verbessert [11].

3.4 Konzepte

Eine andere Definition von MDSD, die [12] zugrunde liegt, lautet: *„Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken die aus formalen Modellen automatisiert lauffähige Software erzeugen.“*

Techniken und Konzepte, die in diesem Zitat angesprochen sind, bilden die Basis für die modellgetriebene Entwicklung bilden und werden in diesem Kapitel genauer betrachtet.

Die folgenden Konzepte werden – falls nicht anders zitiert – aus den entsprechenden Kapiteln von [12] zusammengefasst.

3.4.1 Metamodellierung

Wie schon eingangs erwähnt, spielen Modelle ein zentrale Rolle in der modellgetriebenen Entwicklung. Diese bestehen aus Daten, welche einer Struktur zugrunde liegen und eine Bedeutung in diesem Kontext haben. Um eine automatisierte Verarbeitung (wie Codegenerierung, Modelltransformation, etc) gewährleisten zu können, müssen Struktur und Bedeutung exakt und eindeutig definiert vorliegen.

Ein Metamodell entspricht der Definition von Struktur und Bedeutung für eine bestimmte Art von Modellen. Auf einer abstrakten Ebene werden Modellkonstrukte, Beziehungen unter den Elementen und Gültigkeitsregeln spezifiziert. Zur Visualisierung kann man die Beziehung von Metamodell zu Modell mit der von Klasse zu Instanz vergleichen: In diesem Fall ist das Modell eine Instanz des Metamodells.

Die Beschreibung eines Metamodells könnte im Prinzip auch textuell vorliegen, aber auch hier macht eine formale Darstellung durch ein Metametamodell Sinn. Diese Schachtelung wäre theoretisch auch noch weiter denkbar, in der Praxis geht man aber nicht über die Metametaebene hinaus. In diesem Fall beschreibt die Metametaebene sich selbst.

Im Standard der „Object Management Group“ OMG taucht z.B. die „Meta Object Facility“ (MOF) als Metametamodel auf, welche sich auch selbst beschreibt. Die Unified Modeling Language (UML) ist eine Instanz der MOF und konkrete UML-Modelle sind Ausprägungen des UML-Metamodells.

[9]Siehe Kapitel 3.4.1 und 3.4.2

3.4.2 Domänenspezifische Sprachen

Model-Driven Software Development spielt sich immer im Kontext von Domänen ab. Diese beschreiben einen eingeschränkt ausgewählten Bereich auf den sie Bezug nehmen (z.B. Versicherungswesen oder Telekommunikationindustrie). Domänen können in technische und fachliche eingeteilt werden, falls dies relevant und gewünscht ist.

Mit dem, im vorigen Kapitel gezeigten, Konzept der Metamodellierung lässt sich abstrakt die Strukur von Domänen beschreiben. Solche Modellierungssprachen werden allgemein als Domain Specific Languages (DSLs) bezeichnet. Ein Ziel von DSLs ist es, dem Benutzer der Sprache so gut wie möglich zu helfen, in einer Domäne zu denken und zu arbeiten[10]. Erst durch die Orientierung am Problemraum (der jeweiligen Domäne) können spezielle DSLs mit kompakter Syntax erzeugt werden.

Domänenspezifische Sprachen können bei Bedarf auch selbst erzeugt und auf eine spezifische Probleme zugeschnitten werden. In diesem Fall sollte als erstes die Domäne analysiert werden. Dabei werden geeignete Abstraktionen und Konzepte gesucht, um diese zu beschreiben und ein Metamodell zu bilden. Nachfolgend kann die exakte Syntax spezifiziert werden.

3.4.3 Zielarchitekturen und Domänenspezifische Plattformen

Softwarearchitektur spielt eine wichtige Rolle im Entwicklungsprozess und wird hier in einem relevanten MDSD-Kontext dargestellt:

- Auch bei der modellgetriebenen Entwicklung wird Softwarearchitektur ganz allgemein eingesetzt, um das zu erzeugende System zu strukturieren. Es ergeben sich die gleichen Fragestellungen und Probleme wie bei z.B. der Modell-basierten Entwicklung. Bei der modellgetriebenen Entwicklung kommt als weiterer Aspekt hinzu, dass von der Softwarearchitektur auf eine Zielarchitektur abgebildet wird, welche auch die Plattformarchitektur umfasst (siehe unten).
- Die Softwarearchitektur hat auch bei der Transformation einen hohen Stellenwert. Durch diesen Schritt

[10]In [12] wird erwähnt, dass bei sehr vielen Projekten UML als grafische Modellierungssprache als Standard verwendet wird und über andere Sprachen nicht nachgedacht wird bzw. diese nicht in Erwägung gezogen werden. Die selbe Aussage wird auch für Programmiersprachen, Plattformen und Frameworks getroffen.

wird die Softwarearchitektur des generierten Produktes, auch Zielarchitektur genannt, festgelegt. An dieser Stelle müssen auch ggf. benötigte Integrationspunkte festgelegt werden, die manuell mit Fachlogik ausprogrammiert werden.

Die Zielarchitektur ist zentral in die modellgetriebene Entwicklung eingebettet, den Ziel ist es, diese automatisiert zu generieren. Nur wenn die Konzepte der Zielarchitektur ausführlich definiert sind und sich systematisch durch Transformationsvorschriften beschreiben lassen, ist eine automatische Generierung möglich.

Der generierte Code baut meist auf Programmbibliotheken, Frameworks, Middleware und Ähnlichem auf. Diese Bestandteile können 3rd-Party-Produkte wie .NET oder J2EE sein oder eigens speziell für einen Generator abgestimmte Produkte. Die definierte Grundlage, auf welcher der generierte Code aufbaut, wird Plattform genannt.

Gibt es eine große Differenz zwischen den Konzepten der MDSD-Domäne und der Plattform, so wird in der Regel die Tranformation komplexer. Daher versucht man mit den Domain Specific Languages (DSLs) sich an die Plattformen anzunähern bzw. umgekehrt, die Plattformen an die DSLs anzupassen. Solche Plattformen werden „domänenspezifische Plattformen“ genannt.

4. BEYOND MDSD

In dem Artikel „Biting the Silver Bullet - Toward a Brighter Future for System Development“ [5] aus dem Jahr 1992 beschreibt David Harel eine optimistische Aussicht auf zukünftige Enwicklungsmethoden für komplexe Systeme. Einige Forschungsarbeiten der folgenden Jahren beschritten diesen Weg und unterstrichen seine eigene Einschätzung.

Diese Erkenntnisse führten zu einem Entwicklungsansatz für komplexe reaktive Systeme, welcher bei einer benutzerfreundlichen Anforderungs-Erfassungsmethodik beginnt. Aus den Anforderungen lässt sich die Verhaltensbeschreibung und in letzter Insanz die fertige Implementation herleiten [6].

In Abbildung 2 wird dieses Gesamtkonzept illustriert. Der Artikel „Executable Object Modeling with Statecharts“ behandelt das System Modell, den daraus synthetisierten Sourcecode und die Verbindungen zwischen diesen Systemteilen. Dieser Kern bildet auch die Grundlage für Entwicklungen im Bezug auf MDSD. Wie in der Grafik ersichtlich ist, würde das Konzept um die Dimension der Anforderungen erweitert.

Interessant in diesem Zusammenhang ist, dass niemals der Begriff „Model-Driven Software Development“ genannt wird, obwohl es sich um Grundlagenentwicklungen auf diesem Bereich handelt.

4.1 Formalisierte Anforderungen

Während des ganzen Software-Lebenzyklus ist es wichtig sicher zu stellen, dass das System macht, was wir von ihm fordern. Daher gilt, stets eine Verbindung zwischen dem System und den Anforderungen zu haben. Die Anforderungen können dabei entweder formal, strikt und genau spezifiziert, informal in natürlicher Sprache geschrieben oder als Pseudocode vorliegen. Da sich die Forschung von David Harel mit Automatisierung auf diesem Gebiet beschäftigt, werden hier nur formale Anforderungen betrachtet [6].

Von Message Sequence Charts (MSCs) zu Live Sequence Charts (LSCs): Message Sequence Charts (MSCs)[11] sind eine weit verbreitete und bekannte visuelle Darstellungsform für die Beschreibung von Szenarien welche die Interaktion von Prozessen bzw. Objekten darstellen. Im Umfeld der objektorientieten Entwicklung werden MSCs sehr oft für die Spezifikation von Anforderungen verwendet. Dieses Konzept hat sich auch in UML niedergeschlagen und ist dort unter dem Begriff Sequenzdiagramme zu finden. Sequenzdiagramme in UML besitzen allerdings eine schwächere Ausdruckskraft als MSCs.

Obwohl MSCs heute weit verbreitet sind, wurden einige fundamentale Punkte in ihrer Spezifikation nicht behandelt. Als Beschreibungssprache für Anforderungen sind alle bekannten Versionen von MSCs zu schwach in ihrer Aussagekraft. Diese Sprache genügt nicht, um Aussagen über das Laufzeitverhalten zu treffen und zwischen zwingend notwendigen und optionalen Ereignissen zu unterscheiden. Ein weiteres K.O.-Kriterium ist, dass es nicht möglich ist, ungewollte Szenarien zu beschreiben. Gerade für reaktive Systeme, speziell solche die in sicherheitskritischen Bereichen eingesetzt werden, sind solche Anforderungen essentiell.

In dem Paper „LSCs: Breathing Life into Message Sequence Charts“ wurden diese Probleme behandelt und eine Erweiterung mit dem Namen Live Sequence Charts (LSCs) vorgeschlagen. Wie der Name schon sagt, geht es bei LSCs um „liveness“. Dies drückt aus, dass erwünschtes Verhalten wie z.B. das Terminieren des Programmes eintreten muss. Der Modellierer kann zwischen „möglichem“ und „erforderlichem“ Verhalten unterscheiden, global – auf der Ebene des ganzen Diagramms, oder lokal – wenn Events oder Bedingungen spezifiziert werden bzw. in zeitlichen Abschnitten des Diagramms. Die als lebendig bzw. heiß (live or hot) bezeichneten Elemente ermöglichen es, Antiszenarien zu beschreiben. Die Anderen, welche auch kalt (cold) genannt werden, unterstützen Iterationen und Verzweigungen. [1]

Im Vergleich zu MSCs bieten LSCs eine viel ausdrucksstärkere Art, visuell Anforderungen im Bezug auf das Verhalten darzustellen. Folgende Aussage von David Harel untermauert den Stellenwert von LSCs im Kontext zur modellgetriebenen Entwicklung: *„Because their expressive power is far greater (essentially that of XUML itself), LSCs also make it possible to start looking more seriously at the dichotomy of reactive behaviour – the relationship between the interobject requirements view and the intraobject system model.“* [6]

4.2 Verifikation und Synthese

Zwischen den formalisierten Anforderungen und dem System-Modell existiert eine automatisierbare Verbindung. Auf der einen Seite kann durch einen Transformationsschritt das aus Klassen- und Zustandsdiagrammen bestehende Modell aus

[11] Es gibt einen Standard für MSCs, welcher auf Empfehlung der ITU herausgegeben wurde *(ITU-TS Recommendation Z.120: Message Sequence Chart (MSC), ITU-TS, Geneva, 1996).*

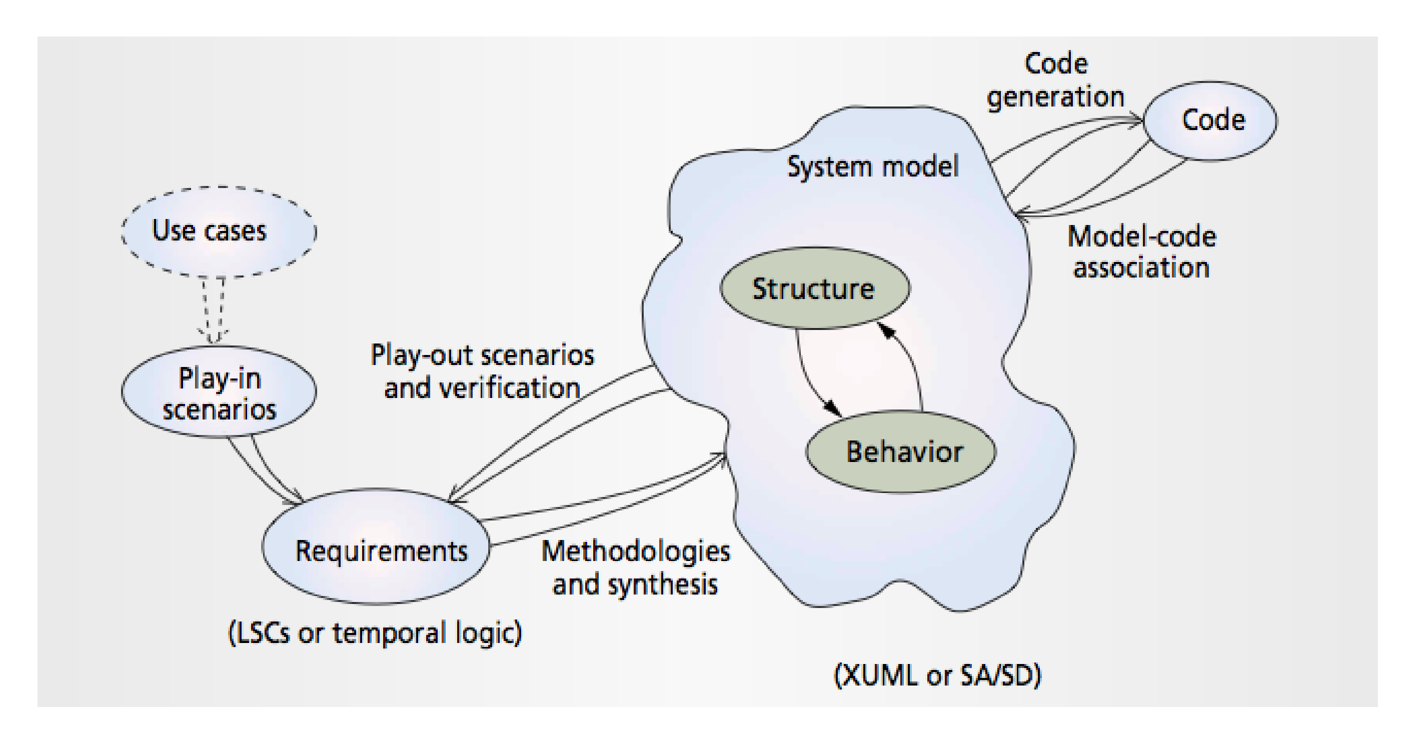


Figure 2: Der Traum der Softwareentwicklung nach David Harel: Von einer komfortablen Anforderungserfassung hin zum ausführbaren Code [6].

den formalisierten Anforderungen generiert werden. Der Weg zurück stützt sich auf mathematische Verfahren, um das Modell gegen die Anforderungen zu verifizieren.

Von den Anforderungen zum Modell: Von den Anforderungen zum Modell bewegen wir uns durch Synthese fort. Hier wird im Gegensatz zu geläufigen Methoden das Modell direkt aus den formalen Anforderungen generiert, wenn diese überhaupt implementierbar sind. In Softwareentwicklungsprozessen, bei denen diese Automatisierung nicht umgesetzt wird, ist es geläufig, dass Entwickler von informalen Methoden geleitet, die Anforderungen manuell in ein Modell umsetzen [6].

Die Generierung des System-Modells aus den Requirements ist ein weitaus schwierigeres Unterfangen, als die Codesynthese aus dem Modell. Die Codesynthes kann auch als „high-level compilation“ angesehen werden, wogegen sich bei der Transformation von den Anforderungen zum System-Modell auch die Frage nach z.B. der Einteilung in Struktur, Objekte und Komponenten stellt [6].

In einem Artikel [8] von D. Harel und H. Kugler, beschreiben sie Ansätze und Verfahren, wie aus Anforderungen welche mittels LSCs spezifiziert sind, Systeme auf Basis von Statecharts generiert werden können.

Vom Modell zu den Anforderungen: Ausgehend vom System Model liegt unser Interesse daran, eine wirkliche Verifikation durchzuführen um das Model gegen die Requirements zu prüfen[12]. Diese Verifikation stützt sich auf mathematische Verfahren um zu beweißen, dass das Modell die Anforderungen erfüllt [6].

Die Verifikation versichert, dass die spezifizierten Antiszenarien niemals eintreten und das explizit geforderte Verhalten eintritt. Im Bezug auf real-time Systeme sei erwähnt, dass auch hier zeitliche Einschränkungen Eingang finden können. Solche Eigenschaften können durch reines Ausführen des Modells nicht sichergestellt werden [6].

4.3 Play-In / Play-Out Szenarien

Mit Play-In Szenarien werden Anforderungen auf der grafischen Oberfläche des zu entwickelnden Systems, bzw. einer abstrakten Version davon, erfasst. Dieser Ansatz ermöglicht das Erfassen von Verhaltensanforderungen abstrahiert von formalen Sprachen auf eine intuitive und benutzerfreundliche Art und Weise [9].

Am Anfang der Entwicklung wird die grafische Oberfläche[13] des Systems erzeugt, ohne ihr zu diesem Zeitpunkt Verhalten zuzuordnen. Aus der Sicht des Benutzers werden die Anforderungen „spielend“ durch die Bedienung der Oberfläche, „drag and drop“ und dem Senden von Nachrichtensequenzen erfasst. In einem zweiten Schritt werden die Perspektive des Systems eingenommen, die Reaktionen auf die Benutzerinteraktion festgehalten und etwaige Bedingungen oder Einschränkungen formuliert. Im Hintergrund werden

[12]Die Begriffe Verifikation und Validation sind abzugrenden von der Bedeutung die sie bei CASE-Tools in den 80er Jahren getragen haben. Damals ging es ausschließlich nur um Konsistenzprüfungen der Syntax bei den Modellen durchführten

[13]Sollte das System keine grafische Oberfläche beinhalten oder es sich um internes Verhalten welches modelliert wird handelt, dann wird bei Play-In Szenarien mit dem

aufgrund der grafisch erfassten Anforderungen die formalen Live Sequence Charts (LSCs) automatisch generiert [3, 9].

Diese Arbeitsweise kommt im Entwicklungsprozess beteiligten Personen entgegen, die a) nicht mit formalen Anforderungsbeschreibungssprachen wie LSCs vertraut sind oder b), sich diese nicht erarbeiten wollen oder einen anderen Zugang zu dem Projekt haben. Zu Letzteren gehören unter anderen, Fachleute und Domänenexperten, Anwendungsentwickler, Anforderungserfasser und eventuell auch potenzielle Benutzer [3].

Die Bestrebungen von Play-In Szenarien zielen ab, auf die Erfassung von so vielen Anforderungen wie möglich durch direktes Manipulieren der grafischen Benutzeroberfläche ohne direkt mit LSCs in Kontakt kommen zu müssen.

Play-Out [9]: Nachdem die Spezifikation mit Hilfe von Play-In Szenarien erfasst wurde, ist es wünschenswert, diese nochmals mit den Forderungen der Kunden abzugleichen und zu verifizieren, damit ein gemeinsames Verständnis vorherrscht. Play-Out erweitert den Ansatz um eine Komponente zum Testen und Validieren von Anforderungen.

Bei Play-Out Szenarien bedient ein Benutzer die grafische Oberfläche des zu entwickelnden Systems gleich, als ob mit finalen Implementierung oder mit einem ausführbaren Modell gearbeitet wird. Beim Bedienen protokolliert die Play-Engine die Aktionen des Benutzers und reagiert selbstständig mit Aktionen und Events, so wie sie bei den Play-In Szenarien erfasst wurden. Ist ein Szenario abgearbeitet, wird dies vom System sichtbar gemacht.

Der große Vorteil liegt darin, dass die Interaktion mit Hilfe von Szenarien schon auf Basis der Anforderungen ausgeführt werden kann, ohne zu diesem Zeitpunkt „ausführbare“ Modelle synthetisieren zu müssen oder einen Prototyp für den Kunden zu bauen.

Das direkte Arbeiten mit der grafischen Oberfläche erleichtert einem breiteren Personenkreis, sich am Prozess beim Überprüfen der Anforderungen zu beteiligen. Durch die Visualisierung und Interaktion ist es nicht unbedingt notwendig sich mit den formalen oder textuellen Spezifikationen auseinanderzusetzen.

David Harel, Hillel Kugler und Rami Marelly sind der Meinung, dass man durch den Play-Out Ansatz gut getestete und verifizierte Spezifikationen bekommt, welche eine geringere Fehlerrate in den folgenden Phasen aufweist.

5. KONKLUSION

Ausgehend von den Erkenntnissen von David Harel und Eran Gery in [7] wurde in dieser Arbeit der Bogen zum Model-Driven Software Development (MDSD) gespannt. Die frühen Konzepte und Ideen der Autoren finden sich heute darin wieder und haben den Ansatz des MDSD mitunter stark geprägt.

Die jüngeren Arbeiten von David Harel gehen einen Schritt weiter in Richtung Formalisierung der Anforderungen und der Möglichkeit, aus diesen das Softwareprodukt zu generieren.

Mit der Arbeit „From Play-In Scenarios to Code: An Achievable Dream“ [6] macht David Harel – wie auch im Titel schon verankert – klar, dass es sich hierbei um eine Traumvorstellung handelt, deren vollständige Realisierung noch länger auf sich warten lassen wird.

Trotzdem gibt es auch heute schon einige Entwicklungsumgebungen, die Model-Driven Software Development unterstützen und damit einen kleinen Markt bedienen. Der Trend nimmt zu und durch die Bemühungen der Object Management Group (OMG) mit der Standardisierung von Model Driven Architecture (MDA) wird dieser, meiner Einschätzung nach, anhalten.

Objektmodell-Diagramm gearbeitet.

6. REFERENCES

[1] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, July 2001.

[2] David S. Frankel. *Model Driven Architecture. Applying MDA to Enterprise Computing.* John Wiley and Sons, 2003.

[3] D. Harel and R. Marelly. Specifying and executing behavioral requirements: The play-in/play-out approach. Technical Report MSC01-15, The Weizmann Institute of Science, 2001.

[4] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[5] David Harel. Biting the silver bullet - toward a brighter future for system development. *IEEE Computer*, 25(1):8–20, 1992.

[6] David Harel. From play-in scenarios to code: An achievable dream. *Lecture Notes in Computer Science*, 1783:22+, 2000.

[7] David Harel and Eran Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, 1997.

[8] David Harel and Hillel Kugler. Synthesizing state-based object systems from LSC specifications. *Lecture Notes in Computer Science*, 2088:1–??, 2001.

[9] David Harel, Hillel Kugler, and Rami Marelly. The play-in/play-out approach and tool: Specifying and executing behavioral requirements.

[10] David Harel and Orna Kupferman. On object systems and behavioral inheritance. *IEEE Trans. Softw. Eng.*, 28(9):889–903, 2002.

[11] Thomas Stahl and Markus Völter. *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management.* dpunkt.verlag, 1. edition, 2005.

[12] Thomas Stahl, Markus Völter, Sven Efftinge, and Arno Haase. *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management.* dpunkt.verlag, 2. edition, 2007.

Aspektorientierte Programmierung – ein Überblick

Rudolf Granitzer
0160067
rudi@edu.uni-klu.ac.at

Abstract

E. W. Dijkstra veröffentlichte 1968 einen Artikel [3], worin er die Struktur des THE Multiprogramming Systems beschrieb. Zum ersten Mal wurde dabei ein Betriebssystem mit einer Schichtarchitektur umgesetzt. Dies erlaubte eine Trennung einzelner Betriebssysstemkomponenten voneinander. Dadurch war es möglich das System Schicht für Schicht zu testen und die Korrektheit der Implementierung des Systems zu zeigen. In modernen Betriebssystemen können sich bestimmte Interessen - wie zum Beispiel höhere Performanz bei Speicherzugriffen oder Logging - über einzelne Komponenten hinweg erstrecken. Es besteht die Gefahr, dass die Prinzipien der Schichtarchitektur durch eine Aufhebung der Komponententrennung verletzt werden. Aspektorientierte Programmierung ist eine Möglichkeit solche komponentenübergreifenden Interessen zu identifizieren und in die Komponententrennung miteinzubeziehen.

1. Einleitung

Edsger W. Dijkstra beschreibt in [3] ausführlich den Schichtaufbau des THE Multiprogramming Systems. Jede Schicht repräsentiert eine Menge von sequentiellen Prozessen, die darüberliegenden Prozessen eine bestimmte Funktionalität anbieten. Dabei entsteht ein hierarchisches System. Ziel ist es, von der Hardware immer weiter zu abstrahieren und das System besser verständlich zu machen. Die einzelnen Schichten von THE können als virtuelle Maschinen betrachtet werden. Dabei ermöglichen wohldefinierte Schnittstellen zwischen den Komponenten eine *use* Beziehung zwischen den Schichten. Ein Prozess einer Schicht verwendet die Dienste, die die darunterliegende Schicht anbietet. Auf unterster Ebene geschieht die Prozesssynchronisierung. Darüber werden Schicht für Schicht Speicherverwaltung, Eingabe-/Ausgabe-Synchronisierung sowie Interprozesskommunikation realisiert. Diese Abstraktion von der Funktionalität der einzelnen Komponenten ermöglicht ein besseres Verständnis des Gesamtsystems ohne verstehen zu müssen wie die Komponenten im Detail arbeiten. Im Folgenden werden kurz die Schichten beschrieben, aus denen sich das THE System zusammensetzt - beginnend bei der untersten Schicht:

Layer 5	The operator
Layer 4	User programs
Layer 3	Input/output management
Layer 2	Operator-process communication
Layer 1	Memory and drum management
Layer 0	Processor allocation and multiprogramming

Abbildung 1. Die Schichten des THE Multiprogramming Systems

Schicht 0: Die unterste Schicht ist für die Prozessorzuweisung zuständig. Hier werden Prozessorzeiten an Prozesse zugeteilt. Es wird sichergestellt, dass kein Prozess einen Prozessor für sich alleine beansprucht. Prioritätsregeln stellen sicher, dass bestimmte Prozesse, wie Eingabe- und Ausgabeprozesse, bevorzugt werden. Auf dieser Ebene geschieht die erste Abstraktion: Prozesse, die über dieser Schicht liegen, müssen sich nicht mehr um die tatsächliche Anzahl der Prozessoren kümmern.

Schicht 1: Diese Schicht nennt Dijkstra „Segment Controller“. Hier geschieht die Speicheverwaltung. Der Inhalt einer Speicherseite wird Segment genannt. Es gibt eine Unterteilung zwischen „Core Segments“, die sich im Hauptspeicher befinden, und „Drum Segments“, die sich im Sekundärspeicher befinden. Der Segment Controller lädt benötigte Segmente vom Sekundärspeicher in den Hauptspeicher und schreibt nicht

mehr benötigte Segmente in den Sekundärspeicher zurück, jedoch nicht zwingend an dieselbe Stelle, an der sie sich ursprünglich befanden. Darüberliegende Prozesse sprechen nun nur mehr Segmente an. Sie müssen sich nicht darum kümmern, ob ein benötigtes Segment sich bereits im Hauptspeicher befindet oder erst vom Sekundärspeicher geladen werden muss.

Schicht 2: In dieser Schicht befindet sich der „Message Interpreter". Dieser kümmert sich um die Zuweisung einer Konsolentastatur, über welche Nachrichten zwischen dem Benutzer und irgendeinem darüberliegenden Prozess ausgetauscht werden können. Der Benutzer muss in der Eröffnungsnachricht angeben, an welchen Prozess seine Konversation gerichtet ist. Beginnt ein Prozess eine Konversation, identifiziert er sich in der Eröffnungsnachricht. Der Name des „Message Interpreter" rührt daher, dass er bei einer vom Benutzer eröffneten Konversation zuerst die Eröffungsnachricht interpretiert, um den Prozess zu identifizieren, an den die Konversation gerichtet ist.

Schicht 3: Hier geschieht das Puffern von Eingabeströmen bzw. das Entpuffern von Ausgabeströmen. In dieser Schicht findet die Abstraktion der eigentlichen Peripheriegeräte statt.

Schicht 4: In dieser Schicht befinden sich die Benutzerprogramme. Auf dem THE System liefen Programme zur Kompilerung, Ausführung und Ausgabe von Algol 60 Programmen.

Schicht 5: Auf der obersten Stufe in dieser Schichtarchitektur steht der Benutzer.

Durch diese hierarchische Architektur ergeben sich einige günstige Eigenschaften, die Dijskstra im Anhang zu [3] anführt:

Ein Prozess einer Schicht kann nur Dienste von Prozessen, die eine Schicht darunter liegen in Anspruch nehmen. Dadurch kann ein Prozess der eine Aufgabe erledigt, nur endlich viele Aufgaben erzeugen, die die Prozesse in unteren Schichten abarbeiten, weil Zyklen ausgeschlossen sind. Die hierarchische Prozessarchitektur verhindert außerdem zyklisches Warten. Das heißt, es kann nicht passieren, dass ein Prozess P auf Prozess Q wartet, der auf Prozess R wartet, der auf Prozess P wartet.

Wiederverwendung ist ein wichtiges Problem in der Softwareentwicklung. In komplexen hierarchischen Systemen ist die Wiederverwendbarkeit der Komponenten ausschlaggebend für die Wartbarkeit. Die Erweiterung moderner Softwaresysteme um neue Features - z.B zur Performanzsteigerung - die mehrere Komponenten verändern, führen meist zu einer Vernachlässigung des Schichtarchitekturkonzepts. Änderungen am Softwaresystem betreffen oft viele Komponenten und die Schnittstellen zwischen den Komponenten werden umgangen. Das führt dazu, dass der Quellcode schwer verständlich wird und die Wiederverwendbarkeit der Komponenten nicht mehr gegeben ist.

2. Aspektorientierte Programmierung

Parnas [8] beschreibt die unterschiedlichen Möglichkeiten der Dekomposition. Die Entscheidung, in welche Komponenten ein Softwaresystem aufgeteilt wird, hat drastische Auswirkungen auf die Komplexität, Wartbarkeit und Erweiterbarkeit des Systems.

Aspektorientierte Programmierung (AOP) ist ein Ansatz, der auf der Objektorientierten Programmierung (OOP) aufbaut und deren Vorteile beibehält.

2.1. Aspekte

Bei der Objektorientierten Programmierung werden einzelne funktionale Einheiten eines Systems identifiziert, die dann zu einem Gesamtsystem zusammengefügt werden. Bestimmte Funktionalitäten eines Systems lassen sich jedoch nicht eindeutig einer funktionalen Komponente zuordnen. So sind zum Beispiel Effizenz bei Speicherzugriffen bzw. Fehlerbehandlung systemweite Interessen.

> „*Oft muss bei Paaren von Concerns entschieden werden, nach welchem der Concerns das System zu modularisieren ist. Der andere Concern, der nicht Grundlage des Systementwurfs war, verteilt sich dann über das gesamte System und bildet einen Crosscutting-Concern.*" [9].

Diese Eigenschaften eines Systems, sie werden auch querschneidende Interessen (Cross-Cutting Concerns) genannt, weil sie quer durch funktionale Komponenten schneiden, werden von Aspekten behandelt. Schon in frühen Stadien des Designs sollte eine klare Abgrenzung zwischen Komponenten und Aspekten geschehen. Ähnlich der Trennung von funktionalen Komponenten sollten auch Aspekte voneinander getrennt werden. Später in der Implementierungsphase werden dann Aspekte und funktionale Komponenten wieder zusammengefügt um ein Softwaresystem zu bilden.

Außerdem helfen Aspekte dabei, den Quellcode-Umfang klein zu halten, wodurch die Wartbarkeit von Software erhöht wird.

Gregor Kiczales et al. [5] stellten das Paradigma der Aspektorientierten Programmierung vor. Anhand kurzer Ausschnitte eines LISP Programmes wurde gezeigt, wie sehr die Umsetzung von *Cross-Cutting*

Concerns in den Quellcode verwoben ist. Komponenten behandeln *Cross-Cutting Concerns* oft unterschiedlich. Wodurch Aufblähung des Quellcodes verursacht wird. Das Beispiel, das in [5] angeführt wird, stammt aus einer Komponente einer realen Anwendung zur Schrifterkennung. Die Komponente besteht aus einfachen Funktionen zur pixelweisen Manipulation von Bildern. Die Implementierung von performanzsteigernden Eigenschaften verursachte bei diesem System einen immensen Anstieg der Komplexität:

> *„The clean implementation of the real system [. . .] is only 768 lines of code; but the tangled implementation, which does the fusion optimization as well as memoization [sic!] of intermediate results, compile-time memory allocation and specialized intermediate datastructures, is 35213 lines. “ [5].*

Die Implementierung einer Anwendung mit Aspektorientierter Programmierung erfordert eine Komponentensprache zur Implementierung der Komponenten, eine Aspektsprache zur Implementierung der Aspekte und einen Aspect-Weaver zur Kombination der beiden Sprachen. *Join Points* sind jene Stellen im Programm der Komponentensprache, an denen die Aspekte eingebracht werden.

2.2. AspectJ

AspectJ [1] ist eine aspektorientierte Erweiterung zu Java und wurde im Xerox Palo Alto Research Center entwickelt.

> *„. . . we have chosen to design AspectJ as a compatible extension to Java so that it will facilitate adoption by current Java programmers. By compatible we mean four things:*
>
> - *Upward compatibility – all legal Java programs must be legal AspectJ programs.*
> - *Platform compatibility – all legal AspectJ programs must run on standard Java virtual machines.*
> - *Tool compatibility – it must be possible to extend existing tools to support AspectJ in a natural way; this includes IDEs, documentation tools, and design tools.*
> - *Programmer compatibility – Programming with AspectJ must feel like a natural extension of programming with Java.“ [4]*

Java wurde als Basis für eine Aspektsprache gewählt, weil sie eine General-purpose Language ist und daher domänenunabhängig ist. Es kann auf dem Modularisierungsprinzip mit Klassen von Java aufgebaut werden, um Aspekte, die quer zu dieser Modularisierung liegen, zu definieren. AspectJ ist sehr gut dokumentiert, dadurch bietet es sich an, um das Konzept der Aspektorientierten Programmierung und die Arbeitsweise eines aspektorientierten Frameworks verständlich zu machen.

In Tabelle 1 sind die dynamischen Join Points aufgelistet, die AspectJ zur Verfügung stellt.

Tabelle 1. Dynamische Join Points in AspectJ [4]

kind of join point	*points in the program execution at which. . .*
method call constructor call*	a method (or a constructor of a class) is called. Call join points are in the calling object, or in no object if the call is from a static method.
method call reception constructor call reception	an object receives a method or constructor call. Reception join points are before method or constructor dispatch, i.e. they happen inside the called object, at a point in the control flow after control has been transferred to the called object, but before any particular method/ constructor has been called.
method execution* constructor execution*	an individual method or constructor is invoked.
field get	a field of an object, class or interface is read.
field set	a field of an object or class is set.
exception handler execution*	an exception handler is invoked.
class initialization*	static initializers for a class, if any, are run.
object initialization*	when the dynamic initializers for a class, if any, are run during object creation.

Im Folgenden wird die Aspektorientierte Programmierung anhand des Beispiels im Artikel von Kiczales et al. [4] beschrieben:

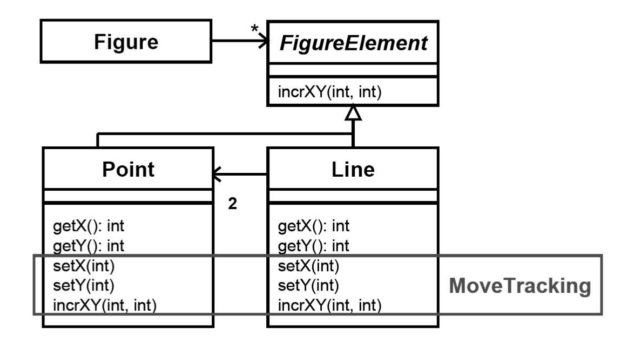


Abbildung 2. Modell eines Figure Editors [4]

Abbildung 2. zeigt das Modell eines Figure Editors. Die Box MoveTracking zeigt einen Aspekt, der zu Methoden in der Line und der Point Klasse „querschneidet".

`Figure` stellt eine Factory dar, die `Line` Objekte erzeugt, die wiederum jeweils aus zwei Punkten bestehen.

```
Point pt1 = new Point(0, 0);
Point pt2 = new Point(4, 4);
Line ln1 = new Line(pt1, pt2);
ln1.incrXY(3, 6);
```

Am Beginn der Ausführung werden zuerst zwei Punkte intialisiert. Danach wird mit diesen zwei Punkten eine Linie initialisiert. Bei der Ausführung der Methode *incrXY()* wird eine Berechnung durchgeführt. Die Join Points, die diese Berechnung durchläuft, sind in Abbildung 3. zu sehen:

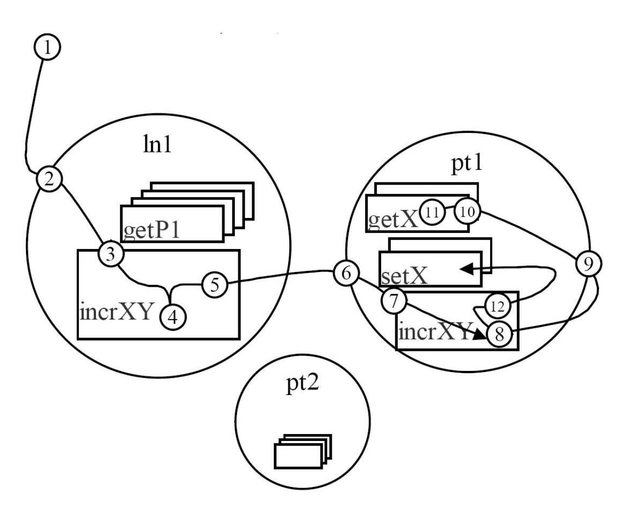


Abbildung 3. Durchlaufene Join Points im Figure Editor [4]

„1. Ein Method Call Join Point, der zur Methode `incrXY` gehört wird im Objekt `ln1` aufgerufen.

2. Ein Method Call Reception Join Point, an dem `ln1` den Aufruf von `incrXY` erhält.

3. Ein Method Execution Join Point, an dem eine bestimmte `incrXY` Methode, die in der Klasse `Line` definiert ist, beginnt mit der Ausführung.

4. Ein Field Get Join Point an dem das Feld `_p1` von `ln1` gelesen wird.

5. Ein Method Call Join Point, an dem die Methode `incrXY` auf dem Objekt `pt1` aufgerufen wird.

   ...

8. Ein Method Call Join Point, an dem die Methode `getX` auf dem Objekt `pt1` aufgerufen wird.

   ...

11. Ein Field Get Join Point, an dem das Feld `_x` des Punktes `pt1` gelesen wird. Die Kontrolle kehrt über die Join Points 11, 10, 9 und 8 zurück.

12. Ein Method Call Join Point, an dem die Methode `setX` auf `p1` aufgerufen wird. ... und so weiter, bis die Kontrolle schließlich über 3, 2 and 1 zurückkehrt" [4] (Originaltext in Englischer Sprache)

2.3. Pointcut Designators

Pointcuts sind eine Zusammenfassung mehrerer Join Points. Pointcut Designators sind Bezeichner dieser Join Points, sie identifizieren Join Points. AspectJ enthält bereits vordefinierte Pointcut Designatiors.

`instanceof` ist ein primitiver Pointcut Designator. `instanceof(FigureElement)` identifiziert beispielsweise alle Join Points, an denen das aktuell behandelte Objekt vom Typ `FigureElement` bzw. eine Unterklasse von `FigureElement` ist. Pointcut Designators können mit den logischen Operatoren `and`, `or`, `not` (&&, ||, !) kombiniert werden.

```
!instanceof(FigureElement) &&
calls(void FigureElement.incrXY(int,
  int))
```

Dieser Codeabschnitt identifiziert alle Join Points, an denen die Methode `FigureElement.incrXY(int, int)` aufgerufen wird und der Aufrufer selbst nicht vom Typ Figureelement ist.

Weiters können benutzerdefinierte Pointcut Designators erstellt werden:

```
pointcut moves():
receptions(void FigureElement.incrXY
    (
  int,int)) ||
receptions(void Line.setP1(Point))
    ||
receptions(void Line.setP2(Point))
    ||
receptions(void Point.setX(int)) ||
receptions(void Point.setY(int));
```

Hier wurde ein benutzerdefinierter Pointcut Designator deklariert, der alle Join Points identifiziert, die durchlaufen werden, wenn Methoden, die Figuren bewegen können, aufgerufen werden.

2.4. Advice

Advice ist ein Konstrukt in AspectJ, das ähnlich zu einer Methode ist, und angibt welcher Code an den einzelnen Join Points eines Pointcuts ausgeführt werden soll.

AspectJ unterstützt *before, after* und *around* Advices. Zusätzlich gibt es in AspectJ noch Spezialfälle des *after* Advice: *after returning* und *after throwing* unterscheiden zwischen dem normalen Zurückkehren aus einem Join Point und dem Werfen einer Ausnahme - das einfache *after* Advice unterscheidet nicht zwischen diesen Möglichkeiten und ist vergleichbar mit *finally* in Java. Ein *around* Advice wird ausgeführt wenn ein Join Point erreicht wurde und kann die Ausführung dieses Join Points verhindern. [2]

Die Ausführungsreihenfolge von Advices ist wie folgt:

1. *around* Advices. Das spezifischste *around* Advice zuerst. Wird innerhalb eines *around* advice `proceed()` aufgerufen so wird mit dem nächstspezifischsten *around* Advice fortgesetzt, oder wenn nicht vorhanden, zu Schritt 2 weitergegangen.
2. *before* Advices. Das spezifischste *before* Advice zuerst.
3. Berechnungen des Join Points werden ausgeführt.
4. *after running* oder *after throwing* Advices werden ausgeführt, weniger spezifische zuerst.
5. *after* Advices werden ausgeführt, weniger spezifische zuerst.
6. Der Rückgabewert von Schritt 3. (sofern es einen gibt) wird an das innerste *around* Advice in Schritt 1. übergeben. Dieses innerste *around* Advice wird weiter ausgeführt.
7. Ist die Ausführung des innersten *around* Advices beendet wird, wird die Ausführung beim umgebenden *around* Advice fortgesetzt.
8. Ist das äußerste *around* advice fertig wird der Join Point verlassen. [4]

Die folgende Advice Deklaration definiert ein `after` Advice auf dem Pointcut `moves()`, und wird ausgeführt wenn ein `move` Methodenaufrufbeendet wurde. Das heißt, wurde eine Figur bewegt wird das Flag auf true gesetzt.

```
after(): moves() {
  flag = true;
}
```

2.5. Aspekte in AspectJ

Aspekte können Deklarationen von Pointcuts, Advices und sonstiger in Klassen erlaubten Konstrukten enthalten.

Die folgende Aspektdeklaration implementiert eine Änderungsverfolgung, die ermittelt, ob eine der Figuren bewegt wurde.

```
aspect MoveTracking {
  static boolean flag = false;
  static boolean testAndClear() {
  boolean result = flag;
  flag = false;
  return result;
}
pointcut moves():
  receptions(void FigureElement.
      incrXY(int, int)) ||
  receptions(void Line.setP1(Point))
      ||
  receptions(void Line.setP2(Point))
      ||
  receptions(void Point.setX(int))
     ||
  receptions(void Point.setY(int));

  after(): moves() {
    flag = true;
  }
}
```

2.6. Aspect Weaving in AspectJ

In AspectJ wird die Strategie gewählt, einen Compiler die Arbeit des Zusammenfügens des Aspekt- und des Komponenten-Codes erledigen zu lassen. Der Compiler ändert nur jene Stellen des Programms, an denen

Advices Anwendung finden. Advices werden als Standardmethoden in den Code eingefügt. An jenen Stellen im Programm, auf die Advices angewendet werden, werden statische Punkte eingefügt, die den dynamischen Join Points entsprechen. In diese statischen Punkte wird Code zur Ausführung der Kontroll Advices eingefügt (*around advices*).

2.7. Sprachunabhängige AOP

Die Vorgehensweise, Aspektorientierte Programmierung an eine bestimmte Programmiersprache zu binden, nennen Lafferty und Cahill [6] Tyrannei. Das Ziel von AOP sei es doch sich von den vorgegebenen Modularisierungskonzepten einer Programmiersprache zu lösen. Die Aspektsprache an eine Programmiersprache zu binden widerstrebt diesem Ziel.

Lafferty und Cahill stellen in ihrem Artikel das Programmiermodell von Weave.NET vor. Weave.NET unterstützt Aspektorientierte Programmierung für die Common Language Infrastructure (CLI) Implementierung Microsoft.NET. Dies ermöglicht AOP für alle Sprachen die von dieser CLI unterstützt werden. Ähnlich zu AspectJ unterstützt auch Weave.NET Join Points, Pointcuts und Advices. Die Aspekte können hier jedoch keine sprachspezifischen Elemente enthalten, wie es bei AspectJ der Fall ist (Aspekt Deklarationen in AspectJ können ja Java Konstrukte enthalten). Das ermöglicht Aspekt-Verhalten einer Sprache auch auf eine andere Sprache anzuwenden.

Aspekte werden in Weave.NET mithilfe einer XML Beschreibungssprache definiert. Aspekte werden in einer separaten XML-Datei gehalten. Der Weave.NET Weaver ist als .NET Komponente implementiert. Er akzeptiert eine Komponentensammlung und eine XML-Datei, die die Definitionen der Aspekte enthält, als Input. Der Weaver fügt die Aspekte auf der Ebene der CLI Intermediate Language (IL) ein und generiert eine neue Komponentensammlung.

Im folgenden möchte ich anhand des Beispiels aus [6], das den Aspekt des Loggings beschreibt (in diesem Fall ein Terminal, das die Eingaben und Ausgaben mitlogt) die Unterschiede zwischen AspectJ und Weave.NET hinsichtlich der Aspekt Deklaration zeigen: Zunächst der Code für die Komponentendeklaration, wie er in AspectJ geschrieben würde:

```
public class Terminal extends
    TerminalConsole
{
  public void WriteLine(String s) {
    Write(s + "'\n"');
  }

  ...
}
```

Nun die Aspektdeklaration zum Logging Aspekt in AspectJ:

```
export System.*
aspect IOChecker {
  pointcut Write(object data):
    execution( * TerminalWrite* (*))
        && args(data);
  before (object data): Write(data)
      {
    LogWrite(data);
  }
  public void LogWrite(object data)
      {
    System.out.println(data.toString
        ());
  }
}
```

Im Vergleich zum vorherigen Code in AspectJ hier nun die Umsetzung in Weave.NET:

Der Code beginnt wieder mit der Deklaration der Terminal Komponente:

```
public class Terminal extends
    TerminalConsole
{
  public void WriteLine(String s) {
    Write(s + "'\n"');
  }

  ...
}
```

Der nächste Codeabschnitt zeigt die Aspektdeklaration in XML (im Unterschied zu AspectJ steht bei Weave.NET kein Aspektverhalten in der Aspektdeklaration):

```
<ax:aspect ... >
<name>IOChecker</name>
<assembly>Aspect_IOChecking </assembly>
<type>Aspect_IOChecking.IOChecking</type>
<body>
 <item><advice><before>
  <formal_param><var_type>object</var_type>
               <var_name>data</var_name>
  </formal_param>
  <pointcut><primitive><pointcutId>
   <name>Write</name>
   </pointcutId></primitive></pointcut>
  <behaviour><name>LogWrite</name></behaviour>
 </before></advice></item>
 <item><named_pointcut>
  <modifier><public/></modifier>
   <name>Write</name>
   <local_var_ref><var_type>object</var_type>
                  <var_name>data</var_name>
   </local_var_ref>
  <pointcut><and>
   <pointcut><primitive><execution>
    <method_signature> ... </method_signature>
   </execution></primitive></pointcut>
   <pointcut><primitive><args>
    <parameter> ... </parameter>
   </args></primitive></pointcut>
  </and></pointcut>
 </named_pointcut></item>
</body>
</ax:aspect>
```

Abbildung 4. Aspektdeklaration in Weave.NET

Das Aspekt-Verhalten wird in C# implementiert:

```
using System;
namespace Aspect_IOChecking
{
 public class IOChecking {
  public void LogWrite(object data)
  {
   Console.WriteLine(data.toString()
       );
  }
 ...
}
```

3. Zusamenfassung und Ausblick

Wenn in einem System *Crosscutting Concerns* identifiziert werden können, macht es Sinn Aspektorientierte Programmierung als Paradigma der Softwareentwicklung einzusetzen. Crosscutting Concerns verursachen Redundaten Quellcode, d.h. gleicher bzw. ähnlicher Quellcode ist über die gesamte Applikation verstreut. Das macht den Quellcode schwer verständlich und schwer veränderbar. Durch Aspektorientierte Programmierung können Kosten bei der Wartung oder Erweiterung des System gespart werden. Eine klare Abgrenzung von Aspekten von den Komponenten eines Systems erhöht die Verständlichkeit des Gesamtsystems. Werden Aspekte richtig identifiziert, dann kann die Aspektorientierte Programmierung stark zur Verbesserung von Wiederverwendbarkeit beitragen. Experimente zeigten, dass Entwickler die AOP verwenden, Fehler in der Applikation schneller finden als jene die das nicht tun. Studien zeigten, dass der Bedarf für Aspektorientierung oft erst bei der Weiterentwicklung von Softwaresystemen auftritt. Neue Anforderungen können sich als Cross-Cutting Concerns herausstellen. In diesem Fall ist es günstig Tools zur Verfügung zu haben die eine Erweiterung des bestehenden Quellcodes um Aspekte ermöglicht. [7] So eignet AspectJ zum Beispiel, um bestehende Java Programme um Aspekte zu erweitern, weil alle gültigen Java Programme auch gültige AspectJ Programme sind.

Dadurch, dass AOP zunehmend in der Lehre vermittelt wird, ist das Aspektorientierte Paradigma dabei die Objektorientierung als State of the Art abzulösen

Literatur

[1] AspectJ PARC Home Page. http://www.parc.com/research/projects/aspectj/default.html.

[2] Introduction to AspectJ. http://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html.

[3] E. W. Dijkstra. The structure of the „THE“-multiprogramming system. *Commun. ACM*, 11(5):341–346, 1968.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[5] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[6] D. Lafferty and V. Cahill. Language-independent aspect-oriented programming. In *OOPSLA ’03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 1–12, New York, NY, USA, 2003. ACM Press.

[7] G. C. Murphy, R. J. Walker, E. L. A. Baniassad, M. P. Robillard, A. Lai, and M. A. Kersten. Does aspect-oriented programming work? *Commun. ACM*, 44(10):75–77, 2001.

[8] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[9] M. Vösgen and D. Sokenou. Aspektorientierte Programmiertechniken im Unit-Testen. *Informatik - Forschung und Entwicklung*, 20(1):57–71, 2005.

The UNIX Programming Environment – Where it Started and Where it Went

Ofner Michael
9969295
crashproof@gmx.at

Abstract

In their 1981 paper "The Unix Programming Environment" [2] Kernighan and Mashey describe the peculiarities of the operating system UNIX and its influence as a programming environment at their current time. Based on this paper, I am going to survey how the aspects they describe as being especially well done or good for productivity still hold up in today's world. Furthermore, I am going to investigate if and how the paradigms that were introduced (e.g. components, pipes and filter) evolved over time and how they compare to today's ideas.

1. Introduction

Computer science and the fields of research associated with it are often considered as quite new and fresh. There is not much literature available on the history of computing and it sometimes seems like we do not want the burden of noting and categorizing historic material. To better understand where we stand today however, we also have to look back and see where we came from. Looking at some of the design choices of earlier days and the problems they had to address back then can open up a whole new view on the paradigms we know today. It is interesting to see if some paradigm was developed without even thinking a lot about it or if it was developed over time, as the solution to a long lasting problem.

In this paper a line is drawn from the development of UNIX and the functions it offers, to some concepts that we commonly know today. While showing these coherences, some additional historic information is given to paint a fuller picture and help to better understand the development which has taken place.

The first chapter of this paper is about the milestones introduced in UNIX and the most interesting parts of the operating system. It describes what UNIX had to offer when it was developed and can be understood as a little historical view from the functionality still available today. On purpose, newer developments in UNIX are not mentioned; there is other literature where this information could be gathered and as described earlier this paper tries to focus on a historical perspective.

Since UNIX was very unique at the time it was developed, it also introduced quite a lot of new programming paradigms and ideas. Some of these ideas like component based development or the pipes and filters architecture are discussed in chapter 3. Additionally to mentioning them and comparing how they deduce from the functionalities described in chapter 2, their use in today's world is shortly examined.

Finally chapter 4 summarizes the most important facts to show how the development of UNIX has influenced the way we produce and think of software today.

2. Milestones of UNIX

Development for UNIX began in 1969 at the AT&T Bell Laboratories. It was designed as designated successor for the Multics operating system which was also invented at Bell Labs but proved to be a failure due to various reasons. UNIX was supposed to eliminate the immanent problems, while some new and interesting features should be added as well. Development was slow at the beginning, since the decision-makers at Bell Labs did not want to risk another loss of money like it had happened in the development of the Multics system. In 1974 the first paper describing UNIX appeared [3]. From this time on, UNIX got widely accepted at AT&T Bell Laboratories and development for it caught on leading to some quite revolutionary concepts being integrated. It is important to note that the ideas for these concepts where not entirely new. The simplicity, with which they could be used and the seaming less integration into the operating system was innovative however.

In 1981 Brian W. Kernighan and John R. Mashey published their paper about UNIX as a programming environment [2]. At this time UNIX was still quite

new. In their paper the two authors described what was revolutionary about the new system and how this had an impact on programming habits.

This paper investigates if these revolutionary ideas are still of interest in today’s world and where development has led them to. To get a perspective on these ideas, the three factors which according to Kernighan and Mashey [2] enabled this development are discussed up front:

1. The file system which is unique in its uniform perspective of the underlying system.
2. A command interpreter with a very capable command language.
3. The very rich set of tools.

2.1. The UNIX file system

The UNIX file system is the cornerstone on which the success of the operating system builds. It has a hierarchical structure and can be represented as a tree with a single root node, where every internal node is a directory and all leaves are either directories or files. Most of the modern file systems are laid out in this way.

What made and makes it special in its way are the design choices taken in its development. Due to these, the file system can be characterized by

- consistent treatment of file data (the kernel hides the internal format in which the data is stored, programs only see an unformatted sequence of bytes),
- dynamic growth of files (file size is only limited by the available storage space),
- the treatment of directories as special files (directories can be read like any other file),
- the treatment of peripheral devices (such as terminals) as files [1].

All of these characteristics have a great effect on the ease of development with the file system but the last one is the most interesting. This idea allows access to peripheral devices with the same programming interface which is also used for regular files. Actually the semantics for reading and writing from/to a device are the same as for a regular file. So in the implementation of most programs no distinction between these two cases has to be made. This leads to a very uniform view on the underlying system which greatly alleviates the development of programs and supports some new concepts as is shown further down.

2.2. The command interpreter

The second factor that according to Kernighan and Mashey [2] led to the widespread use of UNIX is its very powerful command interpreter (the shell). It does not only allow easy execution of programs but it also introduced some interesting concepts which are discussed in this chapter.

The shell itself is a user program and enjoys no special system privileges [5]. This allows easy adaptation of the shell itself to meet particular requirements that may be imposed by a certain use case. The standard shell, as it is delivered with UNIX, is so powerful however, that it would not need a lot of changes for most tasks.

As mentioned above the shell allows the execution of a program by stating its name and the parameters that should be passed to it. If the parameters are files, pattern matching characters are supported (e.g. “*” which stands for any sequence of zero or more characters). These characters are expanded to all files they apply to, by the shell itself. Thus the called program is not affected with the expansion. It only has to offer means “to handle a list of files even if the elementary action performed applies to only one file at a time. [5]“

2.2.1. Standard I/O

Usually a program has to either actively open an existing file or create one to obtain a file descriptor and use it. For most programs however user input as well as output to the user is so common that the shell opens two standard file descriptors at every execution of a program (STDIN, STDOUT). These two descriptors are passed to any program executed by the shell. Since all I/O devices are represented as files, it is easy to accomplish this by pointing the standard input (STDIN) to the file representing the typewriter and the standard output (STDOUT) to the file representing the screen.

It is apparent how the uniform structure of the operating system supports this approach. But the idea goes even further as will be shown in the next chapter.

2.2.2. I/O redirection

As described in the previous chapter, programs handle user input and output through two standard file descriptors which are provided by the shell. Because the UNIX file system offers such extent of abstraction, STDIN and STOUT are handled like any other file. This led to the idea that input could come from and output could go to any other file.

The idea itself “like much else in UNIX, was inspired by an idea from Multics. Multics has a rather general IO redirection mechanism embodying named IO streams that can be dynamically redirected. [4]” In Multics one has to enter a clumsy sequence of commands to acquire such behavior, which states a problem for the common day to day use. In UNIX a simple “>” character is enough to redirect the output of a program to any file whereas “<” redirects a file to the input of a program (Listing 1).

```
PROGRAM < IN > OUT
```

Listing 1: I/O Redirection [2]

This technique opens a set of new functionalities. One, worth looking at, is the idea to origin the input of the shell itself from a file. Since the shell is nothing else but a regular user program which itself reads its input from STDIN this input can be redirected as well. By doing so, one can pass a file to the shell, which contains a sequence of commands. These are then interpreted by the shell (Listing 2). Basically this is the way to write a shell procedure, store it in a file and then execute it. To make this even simpler a file that is marked as executable and contains text is automatically interpreted by the shell if it is executed, without needing the indirection over I/O redirection. By doing so a shell procedure gets indistinguishable from a regular program.

```
SH < CMDFILE
```

Listing 2: Input Redirection with the shell

Another interesting feature offered by I/O redirection is program connection. By directing a file which was the output of a program to be the input of another program, the second program gets dependent on the first one.

```
PROGRAM1 > TEMPFILE
PROGRAM2 < TEMPFILE
```

Listing 3: Program connection by I/O redirection

Listing 3 shows how the output of PROGRAM1 is redirected to a temporary file. In the next step this temporary file serves as input for PROGRAM2 thus connecting the two programs through the temporary file.

2.2.3. Pipes

The preceding chapter showed how two programs can be interconnected to achieve some greater functionality. “It seems silly to have to use temporary files to capture the output of one program and direct it into the input of another. The UNIX pipe facility performs exactly this serial connection without any need for a temporary file. [2]”

```
PROGRAM1 | PROGRAM2
```

Listing 4: Program connection by a pipe

Through this piping technology any programs that hold the convention of reading STDIN and writing to STDOUT can be connected by a pipe. To yield a meaningful overall output though, the output of the first program must have a structure recognizable by the second program. It is important to note that the programs involved do not have to know or care if they are currently used in a pipe. Like described in chapter 2.2.2, programs can be written in a standard way of reading from a standard input and writing to a standard output, all other functionality is provided through the file system or by the shell.

But pipes are more than only an abbreviation for program connection by I/O redirection. Programs in a pipe run concurrently so that PROGRAM2 could consume content and produce output while PROGRAM1 is still itself reading from its input (see Listing 4). Since communication between the two programs is handled through a buffer and PROGRAM2 is emptying this buffer whilst PROGRAM1 is still writing to it, an infinite number of bytes can be sent between them.

In chapter 3.2 I am going to further investigate this concept of pipes which is believed to be one of the major milestones, if not the major milestone, achieved by UNIX.

2.2.4. Command language

The previous chapters showed some of the major features the shell has to offer. The most important among them is the possibility to connect programs through a pipe (see chapter 2.2.3). There is much more functionality in the shell however. Actually it incorporates a command language that offers most of the functionality of fully fledged programming languages like variables, control flow, subroutines and even interrupt handling [2].

By using the very capable command language and the functionality of pipes together with the rich toolkit (chapter 2.3) UNIX provides, very complex programs can be written with low cost and quite good reliability [8]. “Although the shell resembles a typical procedural language, it has rather different qualities. Most important, in certain ways it is a very high-level language, and as such is far easier to learn, use, and understand than lower level languages. [2]” This assumption is mainly based on the fact that a lot of the required functionalities are available in the toolkit and the programmer only needs to make the connection between them.

2.3. The Toolkit

Together with UNIX comes a rich set of tools that covers a lot of the functionality needed in everyday computing tasks. A lot of these tools are written in C but any other programming language that can be compiled to a UNIX program could be used. The fact that they are user programs and not systems programs allows for easy replacement. Thereby performance enhancements or special adaptations can be easily managed.

“Unix tools are built following a set of simple design principles:

- Each tool is responsible for doing a single job well.
- Functionality should be placed where it will do the most good.
- Tools generate textual output that other tools can use – for example, the program output won’t contain decorative headers and trailing information.
- Tools can accept input generated by other tools.
- The tools are capable of performing standalone execution, without user intervention. [9]”

Following these design principles, which are easy to adopt, provides an interface (textual) between the different programs. Using this interface and the UNIX pipe mechanism, new programs can be created that perform almost any function imaginable. In essence these tools are components that can be put together to achieve a higher goal.

Since text output of every tool can not be standardized in a way that it fits as input for any other program, some tools were developed that transform text in various ways. Using these tools the output of the first program can be adapted to fit the input requirements of the second one.

3. Programming Paradigms

As mentioned earlier UNIX was a precursor to many programming paradigms that are common today. UNIX thereby did not really introduce all new ideas but made concepts which where known before so easy to use that they experienced widespread use. This chapter provides a closer look at some of these concepts, models them like we know them today, and gives some insights into their historic development as far as this is possible.

3.1. Component based development

The development of software is a cost and resource intensive undertaking. Many approaches have been tried to overcome this problem. One of which is the idea to split up monolithic programs into smaller generally usable parts and reuse them in the development of other software. By reusing these software components, not only the productivity can be enhanced but also the quality [15].

It is interesting to note that Doug McIlroy who worked at the AT&T Bell Labs at the time Multics (the UNIX predecessor) and UNIX itself were developed was one of the great advocates of component based development and based a lot of the problems of the current slow and unreliable software development on the fact that not enough or no component based development was used [14]. At the same time McIlroy was somehow the inventor of the UNIX pipe functionality as will be explained in chapter 3.2.1.

Is the development with UNIX based tools and the use of pipes as it is described by Kernighan and Mashey [2] component based development or is it just a predecessor to this idea, are questions discussed in this chapter.

3.1.1. Defining a software component

Defining what a software component is is quite difficult. There are many different perceptions on the subject depending on the field of work and education one is coming from. Szyperski offers a definition that captures the subject quite well and gets a lot of citations too: “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. [17]”

From this definition four criteria can be extracted that a software component should fulfil:

- Multiple use: A software component should be usable in different software projects. UNIX tools in this sense are used over and over again. They have multiple usages in the same shell procedure as well as in different programs.
- Context independence: A component should fulfil a certain task and should be able to perform this same task in many different contexts. This criterion is fulfilled as well as the first one since UNIX tools have hundreds of possible usages and are not bound to any context.
- Composition by third parties: A software component should not necessarily be developed in house. It should be possible to develop systems that are building up from parts of many different vendors. UNIX tools fulfil this need in any possible view. First of all they are actually developed from many different people that contributed to the total project. This alone would probably fulfil the stated criterion but UNIX tools are even more. By offering the possibility to develop them in any language that can be compiled under UNIX they offer third parties the opportunity to use the programming language that fits their needs best. As long as they stick to the interface regulations, which basically means ASCII output, the command language can glue them together to a greater whole.

- Independent deployment and versioning: It should be possible to update and change existing components one at a time. Thus making it easier to manage and maintain such systems. Until now it should be clear that the UNIX toolset is doing just that. Since all tools are user programs, they can be changed easily and by anybody. Tools can even be replaced for individual users. Deployment is quite easy too since the program only has to be copied to the right path in the file system.

Given the definition of a software component and comparing it to the functionality the UNIX command language provides together with the extensive toolset (chapter 2) I have the courage to name shell programming a component driven development approach. I am not alone with this assumption as some publications see the UNIX shell as a fourth generation programming language [8] with a component driven approach.

3.2. Pipes and Filters

"One of the most widely admired contributions of Unix to the culture of operating systems and command languages is the pipe, as used in a pipeline of commands", said UNIX co-developer Dennis M. Ritchie in a paper when he looks back on the development process and describes some of the decisions made [4]. This idea of pipes evolved over time and many other systems were built following these principles finally leading to an architectural pattern [12].

3.2.1. History and UNIX pipes

In an internal Bell Labs memo by Doug McIlroy (scanned and put on the net by Ritchie D.M.) written in 1964 he proposes an interesting idea when he lists some of his concerns on a new operating system:

"To put my strongest concerns into a nutshell:

1. We should have some ways of connecting programs like garden hose — screw in another segment when it becomes necessary to message data in another way. This is the way of IO also. [10]"

This new operating system mentioned was the predecessor of UNIX named Multics for which development started around the time this note was written. Even though I/O redirection (chapter 2.2.2) was possible in Multics the possibility of connecting two concurrently running programs without the need of a temporary file was not. But UNIX was not the first to integrate pipes either. The Dartmouth Time-Sharing System as described in 1971 [11], allowed to do pretty much the same things possible with UNIX pipes. However it was not as easy to use and was not exploited to such an extend [4].

In UNIX, pipes did not appear until 1972. Actually, at first it was not even planned to integrate such a facility. Through the insistence of McIlroy however, it was finally added. At the same time the tools, which were already developed and running, were changed to meet the new design principles (see chapter 2.3). From there on the common use of the UNIX pipelining facility has incredibly grown. [4]

3.2.2. Pipes and filters design pattern

"As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. [12]" To solve this problem there has to be a way to adhere models of successful solutions to get back to them and build other systems on the same architectural idea. To do so the notion of architectural patterns was born.

An architectural pattern is a very great abstraction of different best practice and working solutions. To make use of this there also has to be a description of the context in which the pattern is used as well as the forces that the pattern tries to resolve.

One very common architectural style is the pipes and filters style. This paper already showed how UNIX supports the pipelining of commands. It is clear that the UNIX pipelining idea and the ideas presented in the pipes and filters architectural style have a lot in common. One could even say that the great success of UNIX pipes builds the cornerstone for the development of other systems following this idea. Ultimately this led to the creation of an architectural style that summarizes all of these systems ideas.

The basic idea is that the system is built out of components that are connected in a certain way. Each component has a set of inputs and a set of outputs. "A component reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order. This is usually accomplished by applying a local transformation to the input streams and computing incrementally. Output therefore begins before input is consumed. Hence components are termed "filters". The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of another. Hence the connectors are termed "pipes".

Among the important invariants of the style, filters must be independent entities: In particular, they should not share state with other filters. Another important invariant is that filters do not know the identity of their upstream and downstream filters. Their specification might restrict what appears on the input pipes or make guarantees about what appears on the output pipes, but

they may not identify the components at the ends of those pipes. [12]" See Figure 1 for an example configuration.

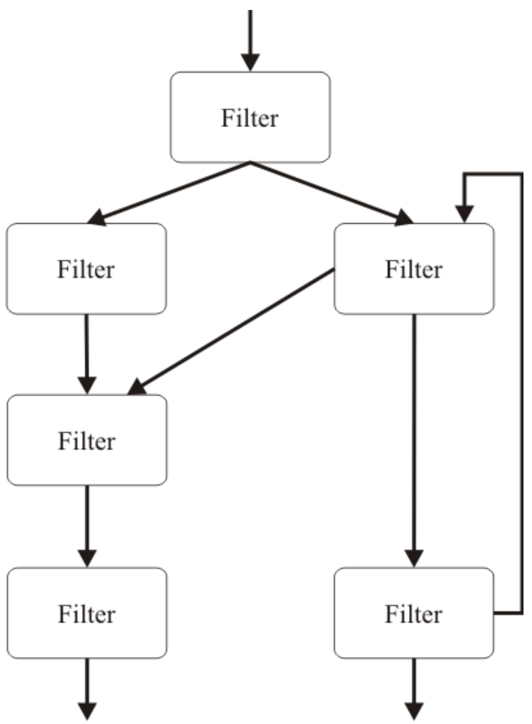


Figure 1: General pipes and filters

If we compare this description of the pipes and filters architecture written by Garlan and Shaw [12] with the pipelining facility that UNIX provides, we see a lot of similarities. In the UNIX world components or filters can be compared to the UNIX tools. Pipes in UNIX offer the same functionality like the ones described in the architectural pattern. At the same time our tools do not know who is their predecessor in the chain or who is coming next, it goes even further as tools can not even tell if they are in a chain or not. There is one major difference though comparing the UNIX pipeline to the general pipes and filters architecture. In UNIX we have a linear sequence of filters with just one input and one output per tool, as shown in Figure 2. In this sense we have a special case, which is mentioned in the architecture description as pipelines or pipelined pipes and filters.

Figure 2: Pipeline

The advantages gained from using UNIX pipes or developing a system based on the pipes and filters architecture are thus pretty much the same:

- First of all it makes it easier for a programmer to understand the I/O behavior by recognizing the overall output as a combination of the individual filters applied. This makes it easier to develop and test such a system. Testing is further enhanced by the ability to look at the preliminary output after any filter and check if it follows the specification until then.
- Another advantage is the fact that this architecture greatly supports reuse of the different filters/components used. This is especially immanent when using UNIX, where the same tools are used over and over again to perform in totally different overall activities.
- Since every filter is independent of any other a system following the pipes and filters architecture can be easily maintained and enhanced. Any component can be replaced by an improved variant [12] and if additional functionality is needed this can be simply done by adding other filters.

There are however also some problems which arise with this kind of architecture. First of all, use of this architecture often leads to a batch organization in processing. Second the coordination between different streams, as it is in the general pipes and filter architecture can be very complicated. Further great care has to be put in breaking up functionality into components. These should be quite low in complexity but at the same time too fine a level of granularity imposes problems too [13]. With the way UNIX implements its pipes another problem arises. All programs in the pipe run concurrently, each in a separate process, which can easily lead to some major performance attrition.

4. Other paradigms

The two paradigms mentioned above are not everything exciting introduced by UNIX. Many other ideas could be derived from this operating system. Some of them are mentioned in this chapter, although they are not discussed as elaborate as component based development and the pipes and filters architecture. Kernighan and Mashey [2] made some interesting observations when looking at the programming methodology that is supported by the shell:

- Programming can often be avoided to a great extent, since the use of readily available tools and the combination of these parts can often fulfill the stated requirements. This idea combines somehow with the idea of component based development (chapter 3.1)

where we also want to reduce programming as far as possible.

- Shell programming can be used to build a prototype for complex solutions before it is implemented in a fully fledged programming language. Since shell programming is more cost effective than normal development such a prototype can be easily thrown away if it does not fit the needs.
- A lot of programs need to be continually modified and live through a kind of evolution until the program settles at a solid state. This is especially true if requirements are continually changing or are not even clear in the first place. A prototype in form of a shell procedure can help greatly in pushing the effort and resources spent compared to the achieved output ratio in a positive direction. Since the shell procedures follow a quite natural language [8] and are built up out of components, changes can be incorporated with relatively low cost and in quite few time.
- When a program has reached its stable state the available shell procedure can be checked for performance gaps and these parts can then be implemented in fully fledged programming languages.

5. Conclusion

The first part of the paper showed how a very well designed file system that offers some quite unique concepts can greatly enhance the flexibility of programs without needing any actual changes to the implementation itself.

After this the UNIX command interpreter – the shell – was discussed. Especially its interaction with the file system is very interesting. Its capabilities are further enhanced with the possibility to connect programs through the use of pipes. This is seen as one of the greatest inventions introduced in UNIX.

The toolbox of programs provided with UNIX together with the command language makes it suitable for a great component based development system at a very high abstraction level.

Even though UNIX was developed a long time ago, it is still in use today and so are its relicts in form of paradigms and ideas. It is important to understand where some of them came from and it is interesting enough to know that they are not that new.

References

[1] Bach M.J.: The Design of the Unix™ Operating System, Prentice Hall, Engelwood Cliffs, N.J., 1986.

[2] Kernighan B.W. and Mashey J.R.: The Unix Programming Environment; IEEE Computer, 1981, pp. 12-24.

[3] Ritchie D.M. and Thompson K.: The Unix Time-sharing System; Comm. ACM 17, No. 7, July 1974, pp. 365-375.

[4] Ritchie D.M.: The Evolution of the Unix Time-sharing System; AT&T Bell Laboratories Technical Journal, Volume 63, No. 6, Part 2, October 1984, pp. 1577-1593.

[5] Ritchie D.M.: The UNIX Time-sharing System – A Retrospective; Bell System Technical Journal, Volume 57, No. 6, Part 2, July-August 1978.

[6] Freitag R.J. and Organick E.I.: The Multics input-output system; Proc. Thid Symposium on Operating System Principles, October 18-20, 1971, pp. 25-41; according to: Ritchie D.M.: The Evolution of the Unix Time-sharing System; AT&T Bell Laboratories Technical Journal, Volume 63, No. 6, Part 2, October 1984, pp. 1577-1593.

[7] Kernighan B.W. and Pike R.: The UNIX Programming Environment; Prentice Hall, London, 1984.

[8] Schaffer E. and Wolf M.: The UNIX Shell as a Fourth Generation Language; Technical Report, Revolutionary Software Inc., Santa Cruz, CA, http://www.rdb.com/lib/4gl.pdf, last visited: 15.05.2007.

[9] Spinellis D: Tool writing: A Forgotten Art; IEEE Software, Volume 22, No. 4, pp. 9-11.

[10] McIlroy M.D.: Internal Bell Labs memo; provided by: Ritchie D.M., http://cm.bell-labs.com/cm/cs/who/dmr/mdmpipe.html, last visited: 15.05.2007.

[11] Systems Programmers Manual for the Dartmouth Time Sharing System for the GE 635 Computer; Dartmouth College, Hanover, New Hampshire, 1971; according to: Ritchie D.M.: The Evolution of the Unix Time-sharing System; AT&T Bell Laboratories Technical Journal, Volume 63, No. 6, Part 2, October 1984, pp. 1577-1593.

[12] Garlan D. and Shaw M.: An Introduction to Software Architecture; in: Ambriola V. and Tortora G. (editors): Advances in Software Engineering and Knowledge Engineering; Volume 1, World Scientific Publishing Company, New Jersey, 1993.

[13] Mittermeier R.T.: Lecture notes for Software-Entwurf, Test und Entwicklungsprozess, Kapitel 5 – Architectural Patterns; Wintersemester 2004/2005 Klagenfurt, 2004.

[14] McIlroy M.D.: Mass Produced Software Components; NATO Software Engineering Conference Report, Garmisch, October, 1968, pp. 79-85.

[15] Mili H., Mili F., and Mili A.: Reusing Software: Issues in Research Directions; IEEE Trans. Software Engineering, Volume 6, No. 6, June 1995, pp. 528-562.

[16] Meyer B.: On to Components; IEEE Computer, Volume 32, No. 1, January 1999, pp. 139-143.

[17] Szyperski C.: Component Software: Beyond Object Oriented Programming; 2nd edition, Addison-Wesley Professional, Boston, 2002.

Software Factory: An Overall Approach to Software Production

Michael Prodnik
0360541
mprodnik@edu.uni-klu.ac.at

Abstract

Der Artikel „A Software Factory: An Overall Approach to Software Production“ steht als ein Klassiker der IT-Literatur außer Zweifel. Die daraus entstandenen Ideen und darin angeführten, bereits seit 1987 gültigen Grundsätze sind auch heute, 20 Jahre später noch genauso aktuell. Unter den zahlreichen beschriebenen Aspekten der Fuchu-Factory können besonders die von Matsumoto herausgehobenen Faktoren der Effektivitätssteigerung durch ein strukturiertes Reuse-Konzept und entsprechende Toolunterstützung gut in die Neuzeit verfolgt werden.

Die Unterschiede der derzeit vorhandenen Gegebenheiten zu den damaligen Vorraussetzungen schaffen durch einen Vergleich von aktuellen Tools und Reuse-Konzepten mit den von Matsumoto geschaffenen Strukturen eine offensichtliche Verbindung zur Gegenwart. Diese Zusammenhänge werden in der nachfolgenden Arbeit durch Untersuchung von aktuellen Tools, Techniken und Standards auf Abhängigkeiten, Zusammenhänge und Verbesserungen gegenüber den damals geforderten Abläufen dargestellt.

1. Einleitung

Matsumoto stellt in seinem Artikel „A Software Factory: An Overall Approach to Software Production“ [1] als Mitarbeiter der Toshiba Corporation in Tokyo, Japan bereits im Jahr 1987 einen schlüssigen und übergreifenden Ansatz zur Softwareproduktion in großem Rahmen vor. Er sieht dabei, der japanischen Kultur entsprechend, Qualität und Produktivität als die zielvorgebenden Faktoren für seine Vorstellung des Softwareentwicklungsprozesses. Von den vielfältigen Konzepten, die großteils den Sprung in die aktuelle Informatik in unterschiedlichen Formen erreicht haben, werden einführend die zentralen Ansätze kurz vorgestellt. Anschließend wird, um das bestehende Bild abzurunden, anhand von Zahlen eine Vergleichbarkeit der damaligen Verhältnisse mit heutigen Gegebenheiten geprüft. Daraufhin werden die Konzepte Reuse und eine technische Workbench als Basis für den Entwicklungsprozess herangezogen, um die Entwicklung und Verwendung der damaligen Überlegungen in der Gegenwart zu beleuchten.

1.1. Das Gesamtbild einer „Software Factory“

Yoshihiro Matsumoto beschreibt seine Erfahrungen anhand der „Fuchu-Software Factory“ in der die Softwareentwicklung in für damalige Verhältnisse höchst industrialisierten, definierten Abläufen durchgeführt wird. Es wird ein strukturierter Entwicklungsprozess mit klaren Verantwortlichkeiten beschrieben. Aus dem Prozess sind bereits klar die unterschiedlichen Phasen

- Planung
- Design
- Implementierung
- Test
- Produktivsetzung des Systems

erkennbar. Benannte „Base Lines“ bilden Review-Zeitpunkte, in welchen nicht nur die Qualität der Produktentwicklung bis zu dem entsprechenden Zeitpunkt geprüft wird, sondern diese auch mit dem persönlichen Profil („Spektrum“) des jeweiligen Mitarbeiters korreliert wird. Somit wird nicht nur der Fortschritt am Produkt, sondern auch mit jedem Prozessschritt der persönliche Fortschritt des Mitarbeiters dokumentiert.

Basierend auf diesen unterschiedlichen Reviews werden die Fehler (erhobene Prozentraten interne Tests / Vor-Ort-Tests) entsprechend in Design Fehler (35/45), Programmierfehler (20/10), Datenfehler (30/20) und Hardware/Interface Fehler (15/25) unterteilt.

Die Strukturierung in unterschiedliche Phasen und eine klare Aufgabenverteilung innerhalb des Erstellungs-Prozesses werden in zwei Bereichen fortgeführt:

Einerseits werden die personenbezogenen Faktoren herausgearbeitet, indem die strukturierten Karrieremög-

lichkeiten der Factory skizziert werden. Es wird die Beurteilung der einzelnen Personen anhand einer Matrix darstellt und der Führungsstil „Look-Forward-Management“ beschrieben.

Andererseits wird die technisch- sowie organisatorisch unterstützende Komponente herausgehoben, indem Matsumoto auch eine Infrastruktur zur Durchführung der notwendigen Arbeiten des Prozesses und eine Software Workbench (SWB) zur informationstechnischen Unterstützung der einzelnen Phasen beschreibt. Insbesondere wird auf das Konzept „Reusability“ eingegangen, dessen Bedeutung mit Zahlen unterstrichen wird: Während 52% der Angestellten bei der halbjährlichen Mitarbeiterbefragung das Reuse-Konzept als entscheidenden Faktor für ihre Produktivität nennen, werden auch die protokollierten Reuse-Raten von Teilen dokumentiert, die zwischen 33% (Reuse von Dokumenten in Designphase) und 48% (Reuse von Programmkomponenten) liegen. Es ist in diesem Bereich bereits die klare Unterscheidung zwischen Software, die für Reuse entwickelt wird und Software die durch Reuse entwickelt wird getroffen, die P.A.V. Hall kurz darauf in [2] näher ausführt.

Als Rahmen für das gesamte Reuse-Konzept wird eine Organisationsstruktur angeführt, in der ein Teil der Arbeitsressourcen ausschließlich mit der Entwicklung und Verwaltung von Reuse-Komponenten beschäftigt ist. Die originäre Programmentwicklung wählt als Kunde dieses hausinternen Dienstleisters Komponenten aus der vorhandenen Komponentendatenbank zur weiteren Verwendung im Programmierprozess der nach außen verkauften Software aus. Um diese Struktur ab einer bestimmten Unternehmensgröße konsequent tragen zu können, koordiniert ein Reuse-Steering-Board als übergeordnete Instanz das Zusammenspiel der zwei Entwicklungsbereiche. Dieses Reuse-Konzept wird aufgrund seiner Bedeutung in weiterer Folge auch bei der Untersuchung der gegenwärtigen Situation detaillierter behandelt.

Diese, bereits stark an moderne Organisationshierarchien erinnernde Struktur wird auf technischer Ebene von einer Datenbank verwaltet, in welcher die Reusekomponenten von einem einfachen Keyword-Retrieval-System unterstützt, gesucht werden können. Zusätzlich wird ein objektorientieres System beschrieben, das auf OKBL (object oriented knowledge based language) basiert. OKBL ist eine objektorientierte Sprache, die von Matsumoto und Yonezawa spezifiziert wurde [3], über die der User durch Interaktion mit den Suchobjekten seine Suchkriterien verfeinern kann und somit zu einem besseren Ergebnis kommt.

Die Entwicklung, welche diese damals etwas mehr als state-of-the-art Sicht der Dinge bis heute genommen hat, wird nun detaillierter untersucht, um anschließend zu sehen, wie sich die beschriebenen Konzepte, insbesondere Reuse und die Software Workbench als Instrument zur technischen Unterstützung entwickelt haben.

2. Evolution in die Gegenwart

Der Vergleich eines 1987 erschienen Werkes mit aktuellen Gegebenheiten kann nicht erfolgen, ohne sich sowohl über diesen Zeitraum entstandene Unterschiede, wie auch Gemeinsamkeiten bewusst zu machen.

Die Softwarebranche hat als zugrundeliegende Basis Hardware, die sich, nach dem über 50 Jahre alten, aber immer noch gültigen Moore'schen Gesetz, alle 2 Jahre in ihrer Komplexität verdoppelt [4]. Aus dieser Ausgangsposition muss als erstes Vergleichskriterium auch bei der darauf aufsetzenden Software Komplexität als erster Maßstab herangezogen werden. Programmiersprachen sind in gleichem Maße komplexer geworden wie die darauf entwickelten Programme. Das durchschnittliche Fuchu-Programm bestand aus 4 Millionen EASL (Equivalent Assembler Sourcelines) was aufgerechnet auf die damals verwendete Haupt-Programmiersprache Fortran ca. 1 Million SLOC (Annahme für Fuchu 1 LOC Fortran = 4 EASL) und eine maximale Programmgröße von knapp 5 Millionen SLOC ergibt. Größere heutige Programme können als Beispiel mit 50 (Windows) – 200 (Debian / GNU) SLOC im Betriebssystembereich angenommen werden [5]. Wir gehen hier zwar nicht mit Moores Gesetz konform, die Steigerung ist jedoch nicht zu übersehen. Die Handhabung dieser gewachsenen Komplexität nach den damals erhobenen Vorstellungen liegt aber, unter Berücksichtigung einer ebenfalls anzunehmenden Verbesserung der technischen Unterstützung, noch im Bereich des Durchführbaren.

Auch die Beschäftigungszahlen der Firmen können als Veränderungs-Maßstab zum Vergleich, insbesondere in den human-orientierten Bereichen von Matsumotos Artikel, herangezogen werden. Fuchu beschäftigte 2.300 Mitarbeiter. Heutige Vertreter der Softwarebranche sind Microsoft mit überdimensionalen rund 76.000 Mitarbeitern [6], Sun mit 34.500 [7], und SAP als deutscher Konzern mit knapp unter 40.000 Mitarbeitern [8]. Dies darf jedoch nicht als exponentieller Maßstab gedeutet werden, da es sich bei besagten Branchenriesen eher um die Ausnahme als die Regel handelt. Die Beispiele unterstreichen aber, dass Softwarehäuser dieser Größe existieren und derartig komplexe Unternehmen auch in Anbetracht der fortschreitenden Globalisierung nicht vernachlässigt werden dürfen. Ein vergleichbarerer Wert eines durchaus großen Unterneh-

mens wäre Google mit rund 3000 Mitarbeitern [9] – ein Maßstab, in welchem das Fuchu-Modell noch einwandfrei funktioniert.

Während die Rahmenbedingungen technischer und humaner Ressourcen für die Entwicklung noch gegeben scheinen, wird sich das von Matsumoto propagierte, der damaligen Zeit angepasste, Schulungs- und Karrieremodell in modernen Unternehmen nicht halten. Die durchschnittlichen fünf bis neun Jahre Grundausbildung, die Fuchu anstrebt, können aus Kostengründen nicht realisiert werden. Einen promovierten Doktor noch fünf Jahre Programmierpraxis sammeln zu lassen erscheint aus didaktischer und integrativer Sicht sinnvoll, in der Praxis ist das Einsteigen von Akademikern in reine Programmierpositionen aber eher selten bis nicht vorhanden. Bereits Diplomanden werden zumeist mit Projektleiter-Tätigkeiten betraut, die Spezialisierung beginnt, bedingt durch die breite Technologiebasis, bereits mit dem Einstieg. Hier ist fraglich, ob es trotz einer Ausbildung, die stärker auf „Angewandte Informatik“ abzielt, als damals, insbesondere aus qualitativer Sicht nicht sinnvoll wäre, einen Schritt zurück zu machen.

Die Rahmenbedingungen sind also gewachsen, aber durchaus noch im Bereich des Vergleichbaren geblieben. Dies sei als Basis für den Einstieg in die tatsächlichen Parallelen und Entwicklungen zwischen dem damaligen Artikel und der Gegenwart genommen.

3. Reuse als aktuelles, elementares Konzept

Anhand der letzten Betrachtungen, den Vergleich rechtfertigend, sollen nun die Erkenntnisse der Fuchu-Factory in ihrer Relevanz für die heutigen Technologien betrachtet werden. Als elementarster Faktor für Effizienzsteigerung wurde von Matsumoto das Konzept Reuse erkannt, das hier in aktuellem Kontext beleuchtet wird.

3.1. Patterns and Practices

Die technischen Möglichkeiten sind enorm gewachsen. Moderne Entwicklungssprachen wie Java oder C# zwingen als objektorientierte Sprachen den Benutzer, sich zumindest in Grundzügen Gedanken über Wiederverwendung zu machen. Dies erfolgt hauptsächlich über den von der Architektur übernommenen Pattern-Ansatz, der sich in seiner bekanntesten Manifestierung als die elementaren Design-Patterns zum Reuse wiederfindet. [10] „Patterns und Practices“ ist als weitere Ausprägung dieser Art immer stärker zu einem Buzzword der Industrie geworden. In „Patterns und Practices“ spiegelt sich auch der bereits damals existente Ansatz wieder, nicht nur Komponenten und deren Strukturen (Patterns), sondern auch Prozesse, einzelne Schritte und Dokumente (Practices) wiederzuverwenden.

Ein interessantes Beispiel aus diesem Bereich in der Moderne ist das bewusst oder unbewusst durch Microsoft auf eben seiner „Patterns and Practices“- Homepage aufgezeigte Verständnis des Begriffes „Software Factories“ in einem anderen Kontext genannt. „Software Factories“ in diesem Zusammenhang bezeichnet ein vorab zusammengestelltes, weitestgehend automatisiertes Verfahren zur Erstellung von Software in einer jeweils konkreten Problemdomain [11]. Dem Entwickler werden dabei Grundaufgaben und Grundüberlegungen anhand der (in diesem Fall nicht allgemeinen, aber vom Hersteller vorgeschlagenen) idealen Parameter abgenommen, vorerstellt und vorkonfiguriert. In diesem Zusammenhang funktioniert die Verwendung des Begriffes „Software Factory“ als gesamtheitlicher Begriff, wie er von Matsumoto verstanden wird, nicht. Es ist aber eine durchaus interessante Detailentwicklung des Reuse-Bereiches, die mit der Meinung von A. Endres (der als IBM Mitarbeiter Schlüsse einer „europäischen Software Factory“, den Böblingen Building Blocks publiziert) konform geht, wenn er die besten formalen Ansätze als solche beschreibt, die schlussendlich automatisiert und in Software gegossen werden können [12].

3.2 Serviceorientierter Reuse

Wesentlich elementarer ist das Reuse-Konzept durch Serviceorientierung, aus technischer Sicht insbesondere WebServices „wiederbelebt“ worden, indem das ursprünglich in Fuchu dargestellte Modell mit dem Internet auf eine breitere Basis gestellt wird. Es wird erstmals Reuse auf einer weltweit einheitlichen technologischen Basis, mit XML als Kommunikationssprache und HTTP als Kommunikationsmittel, über alle Betriebsgrenzen hinweg propagiert. Verglichen mit der von Matsumoto beschriebenen Struktur der Wiederverwendbarkeit bleiben die Faktoren „Auffinden von Komponenten“, „zentrale Datenbank“ und „spezialisierte Entwicklung von Komponenten“ erhalten. Die Komponentenauffindung erfolgt über UDDI (Universal Description Discovery and Integration) durch Keyword-Beschlagwortung. Der Discovery-Prozess erfolgt in stark strukturierter Datenbank-Form. Die verwendeten Termini „Telefonbuch / Pages“ in den UDDI-Strukturen weisen dabei deutlich auf eine listenorientierte Struktur hin. Der von Matsumoto angestrebte, objektorientierte Ansatz zur Informationsfindung bleibt größtenteils auf der Strecke, obwohl er gerade in einem

derartig großen Scope als sinnvoll erachtet werden kann.

Durch eindeutige Schnittstellendefinitionen in WSDL und Implementation des SOAP-Protokolls als Transportstandard, wird der Wiederverwendbarkeit in unterschiedlichen Technologien als Key-Kriterium Rechnung getragen. Zentrale Probleme, die sich aber aus Betrachtungssicht der Fuchu-Factory ergeben und durchaus kritisch hinterfragt werden müssen, sind die Faktoren Kontrolle und Qualität. Während die klare Trennung zwischen wiederverwenbarer Komponente und verwendendem Programm gegeben ist, fehlt im Gesamtbild sowohl das „Steering Board“, als auch eine entsprechende zertifizierende Instanz, welche die Komponenten als verwendbar/getestet einstuft. Im Rahmen einer Organisationsdomain ist dies zwar über interne Richtlinien denk- und auch durchführbar, die Standardsammlung WebServices lässt aber technische Rahmenbedingungen hierfür vermissen. In WSDL oder UDDI sind die Metainformationen über Komponente und Hersteller enthalten, Zertifizierung und Authentifizierung sind aber über diese Standards nicht möglich. Hier bleibt es dem Verwendenden selbst überlassen damit umzugehen. Das Konzept von Reuse als gesteuerter Prozess kann als unzureichend erfüllt angesehen werden.

4. Tools zur Veranschaulichung

Um den bereits beschriebenen, gestiegenen Anforderungen der heutigen Zeit Rechnung zu tragen, bedarf es entsprechend starker Tool-Unterstützung. Matsumoto beschreibt 1987 die für den Entwicklungs-Prozess eingesetzte SWB als ein System aus 6 Hauptkomponenten:

- SWB-I zur Unterstützung des eigentlichen Programmierens/Kompiliernes von Programmen
- SWB-II als Test-Komponente
- SWB-III als Design-Komponente
- SWB-IV zur Unterstützung der On-Site Maintainance von Software bei Kunden
- SWB-P zur Projektmanagement-Unterstützung
- SWB-Q als eigenes Quality-Assurance Tool

Zusätzlich werden noch kleinere Komponenten zur Software-Konfiguration sowie die bereits im vorhergehenden Kapitel beschriebene Komponente zum Support von Reuse-Entwicklung beschrieben. Die technische Basis ist somit klar auf die Unternehmensstruktur abgestimmt und ermöglicht eine eindeutige Durchführung des definierten Prozesses. Ein Vergleich mit aktuellen Programmen beleuchtet näher, inwieweit einerseits die damaligen Programmkomponenten weiterhin existieren und andererseits, ob basierend auf diesen neuen Technologien, eine Führung der Software Factory, wie von Matsumoto beschrieben, im heutigen Umfeld möglich wäre. Zum Vergleich werden zwei gängige Entwicklungsumgebungen in ihrer jeweils aktuellen Version herangezogen: Visual Studio Team System (Microsoft) und Rational Software Delivery Platform (IBM).

4.1 Visual Studio Team System als Tool

Microsoft verfolgt mit Visual Studio Team System [13] das Ziel, seine vielfältigen Entwicklungstechnologien unter einer zentralen Plattform zu vereinigen, die einen Datenbankserver (SQL Server 2005) als Verwaltungszentrale im Hintergrund zur Verfügung hat. Dabei werden die auf das System zugreifenden Personen grundsätzlich in drei Kategorien eingeteilt: Software Architekten, Entwickler und Tester. Für jede dieser Rollen wird eine eigene Umgebung zur Verfügung gestellt, welche die entsprechenden Client-Tools enthält. Auf zentraler Server-Ebene („Team Foundation Server“) finden sich die Komponenten der Bereiche: Build- und Changemanagement, Work Item Tracking, Reporting und Projektmanagement als den Prozess steuernde und unterstützenden Komponenten.

Architekten können im Bereich System Design (auf oberster Ebene), Application und Database Design (auf unterer Ebene) und am Design des Deployment Prozesses jeweils spezialisiert tätig sein. Software Entwickler erhalten durch Code Analyzer (statisch/dynamisch) sowie einen Code Profiler Unterstützung. Die Testing-Tools enthalten Komponenten zum Load-Testing, Manual-Testing, Unit- und Code Coverage-Testing, sowie eine Unterstützung im Test-Case Management.

Wird dieser Aufbau mit der Fuchu-SWB verglichen, so fallen auf den ersten Blick die großen Ähnlichkeiten in der Rollenverteilung auf. Die Bereiche lassen sich im Einzelnen, wie folgt, in den Fuchu SWB-Bausteinen wiederfinden: SWB-I entspricht dem Toolset „Developers“, SWB-II hat als Pendant das Toolset „Testers“, das im Team System besonders stark ausgeprägt ist. SWB-III findet sich in den Teilbereichen System-, Application- und Databasedesign wieder, SWB-IV scheint mit seinen Maintainance Aufgaben nur teilweise in dem Produkt auf - es wird zumindest die Teilaufgabe Deployment als Designer-Aufgabe unterstützt. Weitere Maintainance wird im Rahmen des Product Cycle nicht vorgesehen. SWB-P wird als Serverkomponente „Project Management“ realisiert, SWB-Q kann als Teil der umfangreichen Test-Suite interpretiert werden. Unter-

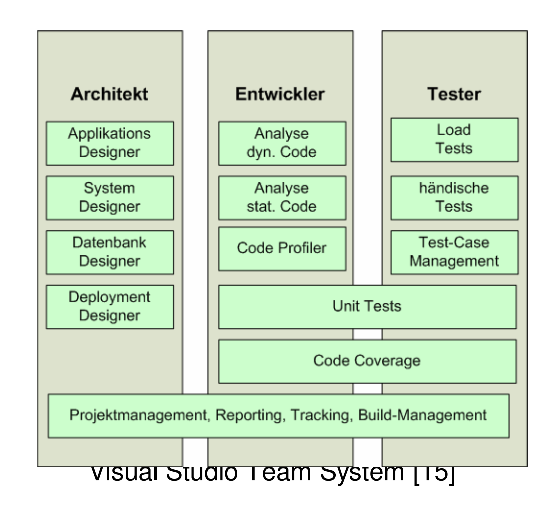
stützung für den Bereich Reuse ist aus der Struktur des Systems nicht ersichtlich und auch bei näherer Betrachtung weder als integratives, noch als administratives Konzept integriert.

4.2 Rational Software Delivery Platform als Tool

IBM führt als formgebenden Prozess für seine „Software Delivery Platform“ [14] RUP (Rapid Unified Processing) an, und unterstellt auf dieser Basis eine iterative Entwicklung. Als an dem Prozess teilnehmende Rollen werden Analytiker, Architekten, Entwickler, Tester, Deployer und als Querschnittsrollen Projekt-Manager und Software Configuration genannt. Im Vergleich mit Matsumotos Ansätzen ist hier bereits auf oberste Ebene eine tiefere Gliederung zu erkennen. In Fuchu werden die Rollen Designer (keine Differenzierung zwischen Analytikern und Architekten) Programmierer, Tester und Wartungspersonal aufgezeigt. Der Delivery Faktor wird vom Tester mitgetragen.

Im Analyse-Bereich bietet die IBM Plattform toolunterstützte Modellierung und Visualisierung von Prozessen, aber auch einen eigenen Bereich zur Anforderungsanalyse und Dokumentation, der den Teamfaktor durch Kommunikationsfunktionen stützt. Verglichen mit der entsprechenden Komponente SWB-III finden sich klar die Funktionen wieder. Der Fokus in der Designphase liegt einerseits, bedingt durch die Orientierung moderner Softwareentwicklung an Geschäftsprozessen und großen Workflowsystemen (SAP, Oracle), sehr stark auf Richtung Business Process Modelling, andererseits ist der Applikations-Designbereich aufgrund der Programmiersprachenvielfalt des anzusprechenden Publikums enorm gewachsen. In der Praxis und zum Vergleichszweck kann aber im Applikations-Design nur das entsprechende, für die Sprache zu verwendende Tool angeführt werden. Der SWB-III Funktion Performance Evaluation können im Produkt von IBM ebenfalls Möglichkeiten zu Performance-Einschätzungen, insbesondere bei unterschiedlichen Systembelastungen, gegenüber gestellt werden.

Das CASAD (Computer aided specification analysis and documentation system), als alle Sichten vereinigendes Programm zur Dokumentenerstellung, findet sich jedoch in solcher Form heute nicht wieder. Es kann hier am ehesten noch mit dem UML-Modellierungstool verglichen werden. Die Sichtweise ist hier jedoch nicht so generalisiert möglich, wie im CASAD Ansatz.



Visual Studio Team System [15]

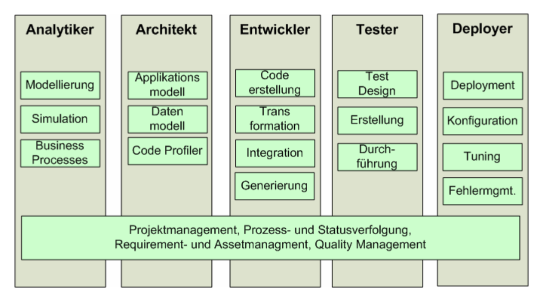


Abbildung 2. Grober Aufbau der Software Delivery Platform [16]

IBM sammelt seine Testing-Tools (SWB-II) unter dem Titel „Software Quality“ und bietet automatisiertes, manuelles, performanceorientiertes und funktionales Testen als Varianten an. Die Bezeichnung „Quality Control“ findet sich in Fuchu wieder, jedoch nicht als integrativer Bestandteil der Test-Suite, sondern abgespalten im eigenen Quality-Control-Support. Dieser ist als übergreifende Funktion nachgelagert, ähnlich dem Projektmanagement, angesiedelt.

Als Querschnittsfunktion, ebenfalls im Toolkit integriert, findet sich die Unterstützung für Software Configuration Management, eine Funktionalität, die von Matsumoto nicht gesondert abstrahiert oder beschrieben wird. Sie ist aber in modernen, kurzen Entwicklungszyklen nicht mehr wegzudenken.

Der separate Deployment-Teil beinhaltet Administrationswerkzeuge für den Datenbank- sowie Webbereich ebenso, wie die weit verbreitet eingesetzte Software-Deployment Lösung „Tivoli“.

5. Lessons Learned

Während im vorhergehenden Kapitel auf die offensichtlichen Parallelen zwischen der Fuchu-SWB und den beiden aktuellen Kandidaten eingegangen wurde, folgen nun die detaillierten Vorteile, die sich aus den Konzepten von Matsumoto ergeben.

5.1 Prozess

Beginnend bei dem straff strukturierten Prozess der Software-Factory kann bereits bei dem Vergleich der abgebildeten Struktur-Grafiken mit den eingangs beschriebenen, einzelnen Phasen des Fuchu-Prozesses eine Abbildbarkeit festgestellt werden. Die Schritte Planning, Requirements Analyse und Design, Extended Design sowie Program Design können in der Microsoft Lösung bis auf den dokumentenorientierten Requirements-Ansatz eindeutig abgebildet und somit die Design-Baseline erreicht werden. IBMs weitergehender Ansatz ist um die Requirements-Analyse reicher und realisiert so den Schritt bis zur Design-Baseline vollständig.

Die Product-Baseline kann von beiden Tools in heutigem Rahmen ebenfalls erreicht werden. Schwierig ist die Beurteilung, ob der strikte, als Konzept sehr interessante Customer-Site Test toolunterstützt ablaufen kann. Matsumoto sieht in seiner SWB dies nicht explizit vor, der Customer-Site Test ist aber integraler Bestandteil seines Prozesses. Vergleichbar hiermit ist eine tatsächliche Simulationsumgebung, die IBM ausgereift für eine Vielzahl von unterschiedlichen Bereichen zur Verfügung stellt, Microsoft zumindest in den Kernbereichen „Web Anwendungen“ und „Windows Anwendungen“ auf niedrigem Level parametrisierbar anbietet. Die IBM-Lösung ist dabei durchaus als Fortschritt anzusehen. Die Auslieferung hat im Rahmen der integrierten Suite nur IBM umgesetzt, Microsoft verwendet zwar einheitliche Standards (MSI Pakete, Microsoft SMS-Server), hat diese jedoch nicht als vollständig durchgängiges Konzept in seinem Toolkit integriert.

Die Operational-Baseline wird von beiden vorgestellten Produkten nicht erreicht. Weder das Zusammenspiel im Echtbetrieb, noch der eigentliche Vor-Ort-Test werden von den Tools unterstützt. Die Möglichkeit zur Unterstützung wäre im stark dokumentengetriebenen Ansatz von Matsumoto zu sehen, der aber generell in den heutigen Tools eher zurückgestellt behandelt wird. IBM spezifiziert Dokumentenverwaltung und Erstellung lediglich im Requirements-Bereich, in Microsoft fehlt die zugehörige Verwaltung vollständig. Es wird davon ausgegangen, dass im Filesystem entsprechende Hierarchien zur strukturierten Ablage existieren/geschaffen werden, oder die Aufgabe an Dokumenten-Management-Systeme übertragen wird. Eine entwicklungsorientierte, dokumentenbasierte Reportingfunktionalität, die in die Suites integriert werden könnte, wäre im durchgängigen Softwareprozess eine Bereicherung, die sich aus Matsumotos Ansätzen mit heutigen Mitteln (Web) durchaus realisieren lassen würde. Über Identifikation aus den Codebereichen können Testdokumente codebereichsspezifisch generiert und über Netzwerke in das entsprechende Entwicklungssystem als Feedback und zu Wartungszwecken rückübertragen werden. Hier eine Infrastruktur zur Verfügung zu stellen würde möglichen Iterationen des Prozesses entsprechend gerechter werden.

5.2 Management

Die von Matsumoto als operative Management-Methode „Look Forward Management“ beschriebene Projektplanung lässt sich mit Hilfe der Tools nur teilweise realisieren. Im Microsoft Produkt lässt sich Projektmanagement anhand aller aufgezeichneten Daten, die aus unterschiedlichen Komponenten zentral am Server aggregiert werden, auf entsprechende Auswertungen gestützt, durchführen. Durch Drilldown-Techniken, mit welchen von zusammenfassenden Berichten in Einzelschritten auf Detaildatensätze „in die Tiefe“ navigiert werden kann, erhält der Manager, basierend auf SQL-Server-Reporting-Services, entsprechenden Einblick auf die von Matsumoto herausgearbeitete menschliche Ebene. Zwar nicht „out-of-the-box“, aber konfigurierbar ließe sich auch ein entsprechendes persönliches Spektrum einer Person erstellen. Die zwei Faktoren „menschliche Beurteilung“ und „Vorauskalkulation“ im Projektmanagementbereich werden aber vollständig vernachlässigt. Sämtliche erhobene und einsehbare Daten basieren auf aus den Einzelkomponenten automatisch generierten Zahlen. Als Beispiel liefern Testkomponenten die erfolgreich durchgeführten Tests/User und Entwicklungs-Tools die fertiggestellten Komponenten/User. Der Faktor einer persönlichen Evaluierung, die ein Mitarbeiter selbst durchführen oder von einem anderen Kollegen erhalten könnte, fehlt vollständig. Damit wird einem Kernkonzept von Matsumotos Prozess, den laufenden Inspektionen, ebenfalls nicht entsprochen. Ein allfälliges Personenspektrum wäre daher strukturell abbildbar, in seiner Aussagekraft aber fragwürdig, da die Daten vollständig automatisiert ermittelt werden würden. Auch der Vorberechnungsfaktor ist nicht berücksichtigt. Es können zwar entsprechende Tasks mit Zeiten vordefiniert werden, Verschiebungen von Zeitplänen werden

entsprechend aktualisiert, das Errechnen von zukünftigen Entwicklungen auf bestehenden Daten ist jedoch nicht vorgesehen, was die Managementtätigkeit unnötig in Richtung Kontrolle beschränkt und den Faktor Planung erschwert. Die Daten für die Planung sind jedoch entsprechend vorhanden (aus Matsumotos Definition eines Workitems fehlt lediglich die erwartete Reuse-Rate als Attribut). Der Kontrollbereich selbst erfüllt damit alle Vorraussetzungen, die Matsumoto an die fabrikähnliche Softwarefertigung stellt, indem von ihm geforderte Workloads pro Einheit oder Person definierbar sind, der Gesamtaufwand entsprechend verfolgbar ist und auch der gegenwärtige Fortschritt sehr transparent abgebildet werden kann.

5.3 Reuse

Die Realisierung des Reuse-Konzeptes ist ein überraschend vernachlässigter Bereich moderner Entwicklungstools. Weder die Komponentenverwaltung in einem eigenen Repository, noch der Prozess einer separaten Komponentenerstellung und Verwendung mit Überwachung kann auch nur ansatzweise in einem der Tools erkannt werden. Der Trend geht eher in die Richtung von Prozess-Reuse. Es werden über Templates allgemeine, wiederverwendbare und parametrisierbare Aufgabenlösungen geschaffen, die dem Entwickler nicht als fertige, getestete und verifizierte Komponente zur Verfügung stehen, sondern ihm nur einen Standard-Rahmen nach Best-Practice Konzept liefern. Der Entwickler parametrisiert diesen und füllt ihn teilweise mit Code, wodurch eine, im Anschluss von Grund auf zu testende Komponente, entsteht. Verglichen mit der gut durchdachten Vorarbeit, die hier in Matsumotos Text und anderen Arbeiten bereits geleistet wurde, ist diese Entwicklung enttäuschend.

6. Lessons taught

Die Sicht auf Matsumotos Artikel umkehrend, soll noch kurz beleuchtet werden, was aus heutiger Sicht im Prozess der Software-Factory zu ergänzen wäre.

Basierend auf den beschriebenen aktuellen Tools kann insbesondere im Deployment Sektor, der bei großen Installationen einen beachtlichen Teil des Aufwandes ausmachen kann, eine umfassende Unterstützung gesehen werden, die Matsumoto in seinen Überlegungen nicht erkennbar ausführt. Der Ansatz von Testing vor Ort geht zwar bereits in diese Richtung, Software aber wirklich automationsgestützt zu verteilen wird nicht berücksichtigt, obwohl er zum industrieorientierten Prozess mit anschließender Logistik passen würde. Dies ist zwar noch für eine gesamtheitliche Sicht einzufordern, mag aber teils das Fehlen auch in der Art der in Fuchu produzierten Software liegen, da bei der Verteilung von Software eher von fertigen, standardisierten, auf viele Rechner zu verteilenden Paketen ausgegangen wird, was bei relativ hardware-naher Prozess- und Kontrollsoftware eher ein untypisches Kriterium darstellt.

Aus aktuelleren Modellen erkennbar und verbesserbar wäre der in Matsumotos Prozessmodell vermisste Ansatz von möglichen Rücksprüngen/Iterationen in einzelnen Phasen. Der Prozess, der sich in seinen Grundzügen sehr dem Wasserfallmodell annähert, ist straff und geradlinig organisiert und eingerichtet. Anhand von modernen Anforderungen und Erkenntnissen, die sich auch in den entsprechenden Prozessmodellen wieder finden, könnte hier entsprechender Verbesserungsbedarf für die Software-Factory abgeleitet werden, der insbesondere im Faktor Qualität seinen Niederschlag finden würde. IBM zeigt dies vor, indem seine Rational Delivery Platform entsprechend auf dem RUP Prozessmodell aufbaut und das entsprechende Wiederdurchlaufen der Phasen/Iterationen unterstützt.

7. Zusammenfassung

Der näher untersuchte Bereich der Tool-Unterstützung im durchgängigen Softwareprozess, der von Matsumoto über das Konzept „Software Workbench" realisiert wurde, zeigt in großen Bereichen eine nur leicht veränderte Übertragung der damaligen Konzepte, sowohl strukturell als auch inhaltlich, in die heutigen Werkzeuge. In den untersuchten Lösungen nicht übernommen wurde entäuschenderweise das hervorragende Reuse-Konzept, während andererseits die realisierten Neuerungen im Bereich der Software-Verteilung durchaus auch Anregungen zur Verbesserung im Fuchu-System liefern. Insgesamt verbleibt der Eindruck, dass die vorgestellten Werkzeuge den provokanten Software Factory Gedanken vielleicht sogar etwas zu weit in maschinelle, automatisierte und strukturierte Erstellung spinnen. Der persönliche Aspekt ist nicht detaillierter zu finden. Er muss vom heutigen Management auf einer zweiten, organisatorischen Schiene abgebildet werden.

Matsumotos Ideen sind für die damalige Zeit sehr drastisch (aufgrund der detaillierten und straffen Strukturierung) und richtungsweisend. Die durch Workbenches unterstützte Prozessstruktur, mit klar definierten Zielen von Qualität und Effizienz, überzeugt als ganzheitliches Konzept durch die Mittel Wiederverwendbarkeit und Vorausplanung. Insbesondere die Ausarbeitung des Reuse-Bereiches, der organisatorisch getragen wird, kann heute noch überzeugen. Das detaillierte

Herunterbrechen von Analysen bis auf ein persönliches Spektrum des Einzelnen überzeugt als Idee zur Qualitätsverbeserung ebenfalls. Eine einfache Umsetzung als rein verbessenders Element, ohne Überwachungsambitionen durchzusetzen oder Paranoia zu wecken, ist in der aktuellen Arbeitswelt jedoch nur in einer gefestigten Firmenstruktur vorstellbar.

8. Referenzen

[1] Matsumoto Y., "A Software Factory: An Overall Approach to Software Production", Freeman P.: Tutorial *Software Reusability*, IEEE.CS Press., 1987, pp. 155-178.

[2] Hall P., "Workshop Overview and Conclusions", Dusink L., Hall P.A.V.(Eds).: *Software Reuse, Utrecht 1989*, Springer Verlag und British Computer Society, 1991.

[3] Matsumoto Y. and A. Yonezawa, "Object Oriented Concurrent Programming and Industrial Software Production", ed E. Erhig etal.: *Proceedings Formal Methods and Software Development*, Springer Verlag, Berlin, 1985, pp. 395-409.

[4] Moore G.E., "Cramming more components on integrated circuits", *Electronics* 38(8), 1965, pp. 114-117.

[5] Wheeler D.A., "More than a Gigabuck: Estimating GNU/Linux's Size", Version 1.0.7., 2002, http:// www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html (Stand 02. Mai 2007)

[6] Microsoft Corporation: "Fast Facts about Microsoft", 2007, http://www.microsoft.com/presspass/insidefacts_ms. mspx, (Stand 02. Mai 2007).

[7] Sun Microsystems., "Company Profile", 2007, http://www.sun.com/aboutsun/company/index.jsp (Stand 02. Mai 2007).

[8] SAP, "The World's Largest Business Software Company", 2007, http://www.sap.com/company/index.epx (Stand 02. Mai 2007).

[9] Google, "Unternehmensprofil", 2007, http://www. google.at/intl/de/corporate/ (Stand 02. Mai 2007)

[10] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "*Design Patterns: Elements of Reusable Object-Oriented Software*", Addison-Wesley Professional Computing Series, Addison-Wesley, Reading Mass. 1994.

[11] Microsoft Cooperation, "Patterns and Practices Developer Center", http://msdn2.microsoft.com/de -de/practices/default.aspx , (Stand 06. Mai 2007)

[12] E. Andres., "Lessons learned in an industrial software lab", IEEE Software 09/93., 1993, pp. 58-61.

[13] Microsoft Corporation, "Team System", http://msdn2.microsoft.com/de-de/teamsystem/ (Stand 10. Mai 2007)

[14] IBM, "Rational Delivery Platform", http://www-128. ibm.com/developerworks/platform/ (Stand 10. Mai 2007)

[15] vereinfachte Darstellung von http://www.ibm.com/ developerworks/platform/ - Fig. 2, (Stand 10. Mai 2007)

[16] vereinfachte Darstellung von http://www.microsoft .com/germany/msdn/vstudio/products/teamsystem/team/defau lt.mspx (Stand 10. Mai 2007)

Komponentenbasierte Softwareentwicklung

Bonifaz Kaufmann
MatrNr. 0361496
bkaufman@edu.uni-klu.ac.at

Abstract

Immer wieder wird vom großen Potential der komponentenbasierten Softwareentwicklung gesprochen. Qualitätssteigerung und Kostenreduktion bilden die Triebfeder dieser Sehnsucht. Doch um das Leistungsvermögen von Softwarekomponenten ausschöpfen zu können, sind wohl überlegte Aktivitäten Voraussetzung. Softwaresysteme zu konstruieren, welche aus wiederverwendbaren vertrauenswürdigen Softwarekomponenten bestehen, erfordert hohe Startaufwendungen. Dieser Artikel zeigt auf, wie die Wissenschaft ihren Teil dazu beiträgt, vertrauenswürdige Komponenten möglich zu machen, indem Dokumentation, Qualität und Zertifizierung zum Gegenstand der Komponentenforschung gemacht wurden. Ebenso wird ein Einblick in die Notwendigkeit von Komponentenmodellen gegeben und ein eigens auf die Komposition von Softwarekomponenten zugeschnittener Softwareprozess vorgestellt.

1. Einleitung

Dass Softwaretechnologie kapitalintensiv ist, scheint seit dem Artikel [31] von Peter Wegner aus dem Jahre 1984 kein Geheimnis mehr zu sein. Wiederverwendbare Ressourcen sind das Kapital der Softwareentwickler und damit Investitionsgüter der Softwareproduktion.

Moderne Entwicklung stützt sich auf eine Vielzahl unterschiedlichster Investitionsgüter. Wegner bringt den Begriff der Kapitalintensität ins Spiel: *„…a production process is capital-intensive if it requires expensive tools or if it involves large startup expenditures“.* Die von Wegner beispielhaft aufgezeigten Softwareaktivitäten [31] lehren uns, dass der Erstellung großer, qualitativ hochwertiger Softwaresysteme kapitalintensive Softwaretechnologie zu Grunde liegt. Dieser Trend hat sich wegen der zunehmend divergierenden Kostenkurven zwischen Hardware und Software, aber auch wegen steigender Softwarequalitätsanforderungen bis heute fortgeführt.

Durch Wiederverwendung können sowohl die Qualität der Software als auch die Produktivität von Softwareentwicklung bedeutend erhöht werden [21]. Softwarekomponenten werden in Wegners Artikel als wichtige wiederverwendbare Ressource hervorgehoben. Auch McIlroy [18] hat bereits 1968 in dieser Technologie großes Zukunftspotential gesehen: „*My thesis is that the software industry is weakly founded, and that one aspect of this weakness is the absence of a software components subindustry“.* Inzwischen ist die Softwarekomponenten-Forschung zu einer Teildisziplin des Software-Engineerings avanciert. Ohne Zweifel lassen sich ähnlich weitreichende Fortschritte auch in den weiteren von Wegner betrachteten Aktivitäten wie Softwareprozesse, Knowledge Engineering oder Programmiersprachen finden. Wegner hat in jeder dieser Aktivitäten eine hohe Kapitalintensität feststellen können. Aus heutiger Sicht spannen die angesprochenen Teilbereiche jedoch ein fast unüberschaubar großes Gebiet des Software-Engineerings auf, weshalb dieser Artikel auf die Untersuchung der Entwicklungen in Bezug auf Softwarekomponenten fokussiert.

Die Softwarekomponenten-Technologie hat in den letzten Jahrzehnten und vor allem seit den frühen 90er Jahren ständig an Bedeutung zugelegt. Im Besonderen weil klar wurde, dass die objektorientierte Technologie den Erwartungen an hoher Wiederverwendbarkeit nicht ausreichend nachkommen kann [25]. Diese Erwartungen sollen nun wiederverwendbare Softwarekomponenten und damit zusammengesetzte Softwaresysteme erfüllen.

Welche Anstrengungen hierzu erforderlich sind, sollen die folgenden Ausführungen verdeutlichen. Kapitel 2 beschreibt unterschiedliche Abstraktionsebenen von Softwarekomponenten und führt gleichzeitig eine Abgrenzung dieses eher vielschichtigen Begriffes der Komponente durch. Darüber hinaus werden einige interessante Weiterentwicklungen in diesem Forschungsgebiet diskutiert, um vertrauenswürdige Komponenten zu ermöglichen. Kapitel 3 beschäftigt sich

mit Komponentenmodellen, welche Interaktionsstandards und Infrastruktur für wiederverwendbare Bausteine definieren. Drei der bekanntesten Modelle werden in diesem Kapitel überblicksweise angesprochen. Kapitel 4 stellt das Paradigma der komponentenbasierten Softwareentwicklung vor. Es werden die Unterschiede zum Wasserfallmodell herausgearbeitet und Stolpersteine wie auch Potential des, auf die Verwendung von Komponenten optimierten, Prozessmodells besprochen. Das letzte Kapitel fasst die wichtigsten Aussagen noch einmal kurz zusammen.

2. Die Softwarekomponente

Über Softwarekomponenten wurde schon viel geschrieben und dementsprechend viele Sichtweisen gibt es auf dieses Thema. Bereits eine Definition fällt schwer, angesichts der Tatsache, dass dieser Begriff erst im Kontext der Verwendung seine wahre Bedeutung zeigt [23][7]. Um ein gutes Gefühl dafür zu bekommen was eine Softwarekomponente darstellt, hilft es, sich die fünf von Meyer [19] vorgestellten Abstraktionsebenen anzusehen.

- *functional abstraction:* Stellt eine einzelne wohl definierte Funktion zur Verfügung, beispielsweise eine mathematische Operation.
- *casual grouping:* Eine Gruppierung ähnlicher funktionaler Elemente oder Datendeklarationen wie sie in C Dateien häufig vorkommen.
- *data abstraction:* Klassen in objektorientierten Programmiersprachen entsprechend dieser Abstraktionsstufe der Datenabstraktion.
- *cluster abstraction:* Frameworks bestehend aus aufeinander abgestimmten, zur Komposition bestimmter Klassen.
- *system abstraction:* Eine weitgehend selbstständig lauffähige Komponente, die in sich abgeschlossen ist und klare Schnittstellendefinitionen besitzt. Beispielhaft wären hier die COTS (commerical off-the-shelf) Komponenten zu nennen.

Obgleich der Komponentenbegriff für jede Abstraktionsart volle Berechtigung hat, so konzentriert sich die wissenschaftliche Diskussion bezüglich Reuse von Softwarekomponenten verstärkt auf die beiden letztgenannten. Auch dieser Artikel untersucht vorwiegend Komponenten mit hohem Abstraktionsgrad, welche im Vergleich zu geringerer Abstraktion den höchsten Gewinn bei Wiederverwendung versprechen. Die Erzeugung als auch Verwendung solcher Komponenten ist, wie wir sehen werden, von vielen kapitalintensiven Aktivitäten und Eigenschaften abhängig.

Die Time-to-Market-Forderung erwartet zügige Entwicklungen bei gleichzeitig hoher Qualität, mit der Eigenschaft schnell auf Veränderungen reagieren zu können. Um diesen hohen Anforderungen gerecht zu werden, lebt schon lange der Traum von leicht zusammen zu setzenden „plug-n-play“ Softwarekomponenten. Die gern zitierte Definition von Szyperski unterstreicht die Richtung dieser Idee.

> *“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”* [28]

Unabhängigkeit. Der Zusatz „third parties“ am Ende der Definition von Szyperski stellt sicher, dass die Nutzbarkeit einer Komponente nicht von Werkzeugen oder Wissen, welche sich nur im Besitz des erzeugenden Lieferanten befinden, eingeschränkt werden soll. Diese Voraussetzung gilt natürlich auch für Komponenten, die nicht von außerhalb kommen. Dies impliziert die Fähigkeit ein Softwaresystem zu erzeugen, welches aus mehreren von unabhängigen Quellen bezogenen Komponenten besteht. Selbst die Assemblierung des Systems durch einen externen Systemintegrator ist damit denkbar [3].

Interfaces. Die Sicherstellung derart flexibler Verwendung von Komponenten bedingt vertraglich spezifizierte und dokumentierte Schnittstellen. Ein Interface (Schnittstelle) definiert, wie eine Komponente angesprochen werden kann und wie sie mit anderen Komponenten verbunden wird. Darüber hinaus legt ein Interface auch fest, welche Operationen von der Komponente angeboten werden. Zu jeder Operation sollten Vor- und Nachbedingungen vereinbart werden. Die meisten gängigen Programmiersprachen verfügen über die Möglichkeit, Schnittstellen explizit zu deklarieren, dadurch wird das Verbergen von Implementierungsdetails unterstützt. Was aber fehlt sind standardisierte Interaktionsprotokolle. Diesem Umstand wird durch die sogenannten Komponentenmodelle Rechnung getragen, welche in einem eigenen Kapitel behandelt werden.

Kontext-Abhängigkeit. Viele Komponenten benötigen zur Bereitstellung ihrer Funktionalität zuerst einmal Input, müssen also beim Aktivieren mit geeigneten Informationen versorgt werden, oder benötigen ihrerseits Dienste anderer Komponenten. Ebenso sind Komponenten implementierungstechnisch meistens an bestimmte Komponentenmodelle gebunden. Diese beiden Bedingungen sollten in einer Komponentenspezifikation keinesfalls fehlen. Diese Form der Anforderungen bezeichnet Szyperski als Kontext-Abhängigkeiten [28].

Blackbox- und Whitebox-Abstraktion. Blackbox- und Whitebox-Abstraktion beschreibt mögliche Sichtbarkeit der Implementierungsdetails von Komponenten. Wird eine Komponente als Blackbox betitelt, so sind lediglich Vor- und Nachbedingungen des Softwarebausteins bekannt. Bei der Whitebox-Sichtbarkeit steht in der Regel der Quellcode zur Verfügung und dem Entwickler ist es möglich jeden internen Arbeitsschritt nachzuvollziehen oder ihn sogar abzuändern. Wird die Möglichkeit der Änderung unterbunden, die Quellcode-Sichtbarkeit aber bleibt erhalten, so spricht man von Glassbox-Sichtbarkeit. Buchi und Weck bieten in [8] eine detaillierte Betrachtung über die Schwierigkeiten der verschiedenen Sichtbarkeitsstufen. Die Autoren haben Lücken bei der Spezifikation von Sichtbarkeiten erkannt. Im Speziellen sind Call-backs in Blackbox-Spezifikationen nicht vorgesehen. Sie schlagen deshalb eine Greybox-Spezifikationssprache als natürliche Erweiterung der Implementierungssprache vor. Demnach ist Greybox-Sichtbarkeit eine ergänzende Spezifikation zur Blackbox-Sichtbarkeit mit der Erweiterung, einige für die Ausführung wichtige Implementierungsdetails sichtbar zu machen. Die unterschiedlichen Sichtbarkeiten sind für das Testen von Komponenten ein entscheidendes Kriterium. Auf diese Problematik wird im Kapitel über komponentenbasierte Softwareentwicklung kurz eingegangen.

Dokumentation. Soll eine Softwarekomponente wiederverwendet werden, ist eine gute Dokumentation des Bausteins eine unumgängliche Notwendigkeit. Gerade wenn ein Systemintegrator eine Applikation aus Komponenten verschiedener Quellen zusammensetzt, kann die Dokumentation spürbar zum Gelingen beitragen. Eine Dokumentation sollte die Selektion und die Validierung der Interoperabilität von Komponenten unterstützen, sowie Auskunft über Qualitätsattribute und Wartbarkeit der Komponenten geben [29]. Noch existiert kein Standardmodell zur Dokumentation von wiederverwendbaren Komponenten. Die meisten Lieferanten haben ihre eigenen Dokumentationsrichtlinien, wodurch die Übersichtlichkeit für Einkäufer von Komponenten erschwert wird.

Die Autoren von [29] haben ein Muster vorgestellt, nach dem Komponenten dokumentiert werden sollten. Ihr Dokumentationsmuster berücksichtigt im Besonderen die Produktlinienentwicklung basierend auf „third-party“ Komponenten. Produkte einer Produktlinie haben in der Regel dieselbe Architektur als Grundlage und bestehen aus unterschiedlich zusammengesetzten Komponenten verschiedener Hersteller. Wie man in Abb. 1 erkennen kann, ist eine nicht unbeachtliche Menge an Informationen für Testzwecke vorgesehen.

Basic Information	**Acceptance information**
• General information • Interfaces • Configuration and composition • Constraints • Functionality • Quality attributes	• Test criteria • Test methods • Test cases • Test environment • Test summary • Test support
Detailed Information	**Support information**
• Technical details • Restrictions • Implementation • Delivery	• Installation guide • Customer support • Tailoring support

Abb. 1: Komponenten Dokumentationsmuster vgl.[29]

Komponenten werden, bevor sie in ein System integriert werden, ausführlich getestet. Dieser Aufwand kann durch eine strukturierte Dokumentation minimiert werden, vor allem, wenn der Lieferant der Komponente testspezifische Informationen beistellt. Für eine genaue Beschreibung einzelner Punkte des Dokumentationsmusters sei auf [29] verwiesen.

Qualitätsmerkmale. Eine gut abgestimmte Dokumentation ist eine qualitätsfördernde Eigenschaft für Komponenten. Qualität ist eine wesentliche Triebfeder für den Erfolg von Softwarekomponenten. Da die Qualität eines Systems von der Qualität des schwächsten Gliedes abhängig ist, wird ein hoher Qualitätsstandard für Komponenten gefordert. Mehrere Forschungsgruppen haben sich deshalb Gedanken darüber gemacht, wie Qualitätskriterien aussehen könnten und wie diese bewertet werden müssten. Bertoa und Vallecillo [4] haben auf Basis des generellen Software-Qualitätsmodells ISO 9126 ein Qualitätsmodell speziell für COTS-Komponenten entworfen. Ein Qualitätsmodell definieren sie wie folgt:

> „*A Quality model is the set of characteristics and sub-characteristics, as well as the relationships between them, that provide the basis for specifying quality requirements and for evaluating quality*.” [4]

Die Benennung der Charakteristika übernehmen sie aus dem ISO 9126 Modell, weisen diesen aber eine neue Bedeutung zu. Die Unterkriterien teilen sie in zwei Kategorien ein. Zum einen Laufzeit-Kriterien, welche während der Ausführung der Komponente beobachtet werden können, und zum anderen die Lebenszyklus-Kriterien, sie können während des Lebenszyklusprozesses beurteilt werden. Jedem dieser Unterkriterien werden nun messbare Attribute zugeteilt. Entscheidend ist, dass Attribute gemessen werden können, denn nur so können Vergleiche zwischen Komponenten sichergestellt werden. Ein allgemein gültiges Qualitätsmodell am Markt zu etablieren, sehen die Autoren als schwierig an. Es kann nicht davon ausgegangen werden, dass Anbieter von COTS-Komponenten aus Gründen des

Images, Marketings oder anderen kommerziellen Veranlassungen, auch negative Qualitätskriterien angeben werden [4]. Daher wird ein Vergleich immer mit Schwierigkeiten behaftet bleiben. Einen ähnlichen Ansatz stellt [2] vor. Ebenso auf ISO 9126 aufbauend, entwickelten diese Wissenschaftler ein Qualitätsframework nicht speziell für COTS-Komponenten, sondern für kleinere, wie die Autoren es nennen „*original*" Softwarekomponenten. Auch Bertrand Meyer stellt in [20] ein Komponentenqualitätsmodell vor. Meyer nennt es das ABCDE der vertrauenswürdigen Komponenten. Das ABCDE steht für die fünf Qualitätskriterien, wobei sich die Buchstaben aus den jeweiligen Anfangsbuchstaben der fünf Kriterien ergeben: *Acceptance, Behavior, Constraints, Design und Extension*.

Zertifizierung. Die oben andiskutierten Qualitätsmodelle sind die Grundlage zur Sicherung von Qualitätskriterien. Evaluiert man nun solche Qualitätszusicherungen, spricht man von Zertifizierung. Die Forschungsgemeinde hat sich seit 1993 mit diesem Thema auseinander gesetzt. Dominierten anfangs noch mathematische und testbasierte Modelle [32] als Evaluierungstechniken, so änderte sich der Fokus um die Jahrtausendwende hin zu Techniken und Modellen, welche sich aus der Vorhersage von Qualitätsanforderungen ergeben [1]. Alvaro et al. untersuchten in „*Software Component Certification: A Survey*" die Forschungsbemühungen seit 1993 und haben dabei zwei Strömungen entdecken können. Die eine Richtung behandelt Zertifizierung vorwiegend von der formalen Seite, also wie die Korrektheit von Eigenschaften bewiesen oder vorhergesagt werden kann. Die andere Strömung geht mehr in Richtung Evaluierung von Qualitätsmodellen und deren Zertifizierung. Die Forschung im Bereich der Zertifizierung von Softwarekomponenten ist nach wie vor aktuell. Boegh [5] weist jedoch darauf hin, dass Fragen wie nun Systeme zertifiziert werden sollen, welche selbst aus zertifizierten Komponenten bestehen, noch offen sind.

Dieses Kapitel sollte die dauerhafte Relevanz von Softwarekomponenten deutlich machen und zeigen, dass Softwarekomponenten auf dem besten Weg sind, qualitativ hochwertige Bausteine für Softwareentwickler zu werden. Viele der aufgezeigten Eigenschaften von Softwarekomponenten waren in Peter Wegners Artikel [31] noch nicht diskutiert. Korrekte einheitliche Dokumentation, Qualitätsmodelle und Zertifizierung von Softwarekomponenten helfen dem von Meyer geprägten Begriff der vertrauenswürdigen Komponenten [20] gerecht zu werden. Darüber hinaus müssen vertrauenswürdige Softwarekomponenten auch miteinander kooperieren können. Sie benötigen zusätzliche kapitalintensive Technologie um miteinander effektiv in Verbindung treten zu können.

3. Komponententechnologie

Die Komponententechnologie stellt die Infrastruktur zur Kommunikation verschiedener, einem System zugehöriger Komponenten sicher. Dazu ist es jedoch nicht alleine damit getan, Interaktionsprotokolle zu definieren und Anfragen entsprechend zu transformieren. Um eine möglichst hohe Transparenz und Vielseitigkeit zu gewährleisten, bieten Komponentenmodelle zusätzliche Dienste wie Namens-, Transaktions-, Ereignisbenachrichtigungsdienste oder auch die Sicherheit betreffende Dienste an. Häufig wird in diesem Zusammenhang auch von Middleware gesprochen, da sie der „Klebstoff" zwischen den jeweiligen Komponenten ist. Einer Marktanalyse von Gartner zufolge hatten Anwendungsintegration und Middleware im Jahre 2005 ein Marktvolumen von 8,5 Mrd. US-Dollars [10]. Emmerich, Aoyama und Sventek [16] möchten auf die weitreichenden Forschungsarbeiten hinweisen, die zu dieser Gewichtigkeit geführt haben. Darüber hinaus geben sie auch einen detaillierten Einblick in die unterschiedlichen Technologien und deren Weiterentwicklung.

Komponentenmodelle sind für sich genommen ein sehr komplexes Thema und tragen nicht unwesentlich zur Kapitalintensität der Softwarekomponenten-Technologie bei, weshalb an dieser Stelle ein einführender Überblick über drei der wichtigsten Komponentenmodelle geboten wird.

CORBA/CCM: Die OMG (Object Management Group), gegründet 1989, hat sich zum Ziel gesetzt Interoperabilität zwischen Objekten zu standardisieren. CORBA (Common Object Request Broker Architecture) ging aus dieser Vereinigung, bestehend aus mehreren hundert Mitgliedsfirmen, hervor. CORBA ermöglicht auf verteilten Objekten Operationen auszuführen, ohne dass der Aufrufende wissen muss wo sich das Objekt befindet, noch in welcher Programmiersprache es implementiert ist. Nicht einmal das Betriebssystem und die spezifischen Kommunikationsprotokolle müssen bekannt sein. Wie bereits erwähnt beschränkt sich CORBA auf Objekte, weshalb die OMG seit der CORBA 3.0 Spezifikation Erweiterungen für die Unterstützung von Komponenten anstrengte. Das dabei entstandene CCM (CORBA Component Model) ermöglicht es Anwendungsentwicklern, Komponenten auf Basis gewöhnlicher CORBA-Dienste zu implementieren, konfigurieren, verwalten und einzusetzen. CCM

bietet vier verschiedene Mechanismen (Ports) an, um Interaktion zwischen Komponenten zu erleichtern [30].

- *Facets:* Geben an welche Schnittstellen von der Komponente zur Verfügung gestellt werden. Somit ist es möglich unterschiedliche Sichten auf die Komponente zu erhalten.
- *Receptacles:* Damit werden Referenzen zu anderen Komponenten bezeichnet. Die Receptacles stellen die Verbindung zwischen den Komponenten sicher.
- *Event sources/sinks*: Diese Ports ermöglichen eventbasierte asynchrone Kommunikation.
- *Attributes:* Dieser Mechanismus wird zur Konfiguration der Komponente genutzt.

CCM ermöglich dadurch plattform- und programmiersprachenunabhängige Kommunikation basierend auf der darunter liegenden CORBA-Architektur. Das Modelldesign von CCM lehnt sich stark an das von Sun entwickelte Komponentenmodell EJB (Enterprise JavaBeans) an, wodurch unter anderem Interoperabilität zu EJB erlangt wird.

EJB: Anders als CORBA ist das auf Java-Technologie basierende Komponentenmodell von Sun zwar plattformunabhängig, aber nicht sprachenunabhängig. Das Konzept dieses Komponentenmodells basiert auf einer offenen Spezifikation des Java Enterprise Edition Servers, des EJB-Containers und verschiedener Komponenten (in diesem Modell als Beans bezeichnet) sowie dem Zusammenspiel dieser Einheiten. Der Container läuft serverseitig als Laufzeitumgebung für Beans. Er überwacht und steuert den Lebenszyklus von Beans, versorgt diese mit diversen Umgebungsinformationen und bietet Dienste wie Transaktionskontrolle, Sicherheitsfunktionalität, Namens- und Verzeichnisdienst sowie einen Nachrichtendienst an. Beans können diese Dienste in Anspruch nehmen. Je nach Aufgabengebiet unterscheidet man bei Enterprise Beans zwischen drei Typen [27]:

- *Session Beans:* Kapseln die Geschäftslogik und stellen Dienste hierfür bereit.
- *Entity Beans:* Diese Beans repräsentieren Datensätze einer Datenbank, werden aber durch die Einführung der Persistenz API in Java EE 5 zunehmend an Bedeutung verlieren.
- *Message Driven Beans:* Sie stellen eine asynchrone Kommunikation zwischen den Beans bereit.

Der am häufigsten genannte Kritikpunkt dieses Modells ist die hohe Komplexität. Mit der kürzlich erschienenen EJB-Spezifikation 3.0 möchte Sun dieser Kritik begegnen. Wichtigste Neuerung ist ein Metasprachkonzept (Annotations), welches die Aufgaben der Entwickler vereinfachen soll. Ziel ist es die Anzahl der zu implementierenden Klassen und Schnittstellen zu verringern. Ebenso sind Vereinfachungen im Umgang mit den einzelnen Beans-Typen Teil der neuen Spezifikation [15].

.NET: Das von Microsoft erstellte Komponentenmodell .NET soll das in die Jahre gekommene COM+/DCOM Modell ablösen. Eine .NET Anwendung wird ähnlich wie bei EJB und dem CCM in einem Container ausgeführt, welcher auch hier für übergeordnete Dienste zuständig ist. Der Container arbeitet in der Regel mit mehrere unterschiedlichen .NET Enterprise Servern zusammen. Beispiele für .NET Enterprise Server können sein: Exchange Server, Application Center, SQL Server, BizTalk oder auch ein Host Integration Server, um Zugriff auf IBM-Mainframe-Applikationen sicherzustellen. Innerhalb des Containers befindet sich ebenso die Geschäftsschicht, bestehend aus sogenannten Managed Components. Sie kapseln die Geschäfts- und Datenlogik, wobei Datenbankzugriffe mittels Active Data Objects (ADO.NET) realisiert werden. .NET bietet darüber hinaus einen einfachen Weg an mittels Webservices über WSDL, SOAP, UDDI und BizTalk mit Geschäftspartnern zu kommunizieren. ASP.NET ermöglicht externen Geräten (Web-Browser, PDA etc.) über Benutzerschnittstellen in Form von HTML mit der Enterprise-Applikation in Verbindung zu treten. Microsofts Komponentenmodell ist weitgehend sprachunabhängig, jedoch für den Einsatz unter hauseigenen Betriebssystemen optimiert [24].

Noch bietet keines dieser Modelle echte Interoperabilität an. Interoperabilität bedeutet die Fähigkeit verschiedenste Softwareanwendungen, geschrieben in unterschiedlichen Programmiersprachen, eingesetzt auf beliebigen Plattformen und Betriebssystemen, auf Grundlage unterschiedlichster Netzwerktechnologien miteinander interagieren und kommunizieren lassen zu können [9]. EJB ist auf die Programmiersprache Java fixiert, .NET auf die Plattform von Microsoft, und Komponenten, die unter CCM laufen sollen, müssen auch erst an dieses Modell angepasst werden. Chiang greift in [9] diese Schwachstelle auf und bietet als Lösung den Einsatz von Adaptertechnologie an. Adapter können die Wiederverwendbarkeit von Softwarekomponenten fördern, da sie nicht mehr nur auf ein Komponentenmodell beschränkt sind. Dennoch erfordert diese Technik weiteren Overhead, und wirkt dem Ziel nach weniger Komplexität und damit auch mehr Qualität entgegen.

Systeme aus wiederverwendbaren Softwarekomponenten zu bauen, erfordert ein speziell auf diese Anforderung zugeschnittenes Prozessmodell. Morisio et al. [22] sehen es als Fehler an, komponentenbasierte Ent-

wicklung ohne Anpassung des traditionellen Prozesses zu betreiben.

4. Komponentenbasierte Softwareentwicklung (CBD)

Seit einigen Jahren hat die komponentenbasierte Entwicklung auch in der Industrie an Bedeutung gewonnen. Die Idee, ganze Systeme aus selbst erstellten oder zugekauften Fertigbauteilen zu entwickeln, verspricht hohe Wiederverwendbarkeit bei steigender Qualität und gleichzeitiger Kostenersparnis. Heineman und Councill sind von der Relevanz des komponentenbasierten Software-Engineerings (CBSE) überzeugt: *„…we believe that CBSE represents the „best practices" in software engineering produced during the last thirty years."* [14]. CBSE befasst sich nach diesen Autoren hauptsächlich mit drei Funktionen: (1) Entwicklung von Software aus vorproduzierten Bauteilen, (2) die Fähigkeit diese Komponenten in anderen Anwendungen wiederzuverwenden und (3) mit der Wartung und Anpassung solcher Teile. Ivica Crnkovic beschäftigt sich eingehend mit dem ersten der genannten Punkte und forscht nach spezifischen Prozessen für komponentenbasierte Softwareentwicklung [11]. Er unterscheidet zwischen Entwicklung von Komponenten und Systementwicklung mit Komponenten. Beide Varianten erfordern unterschiedliche Prozess- und Lebenszyklusmodelle, welche aber durchaus auch nebeneinander ausgeführt werden könnten. Werden Komponenten von „third-parties" produziert, ist dies auch die Regel.

Komponenten-Entwicklungsprozess. Die Entwicklung von Komponenten kann weitgehend mit den klassischen Vorhergehensweisen stattfinden. Besonderer Fokus ist jedoch auf die Komponenteneigenschaft der Wiederverwendbarkeit und damit auf Generalität und Flexibilität zu legen. Daraus ergibt sich ein erhöhter Aufwand für Spezifikation und Test. Die Komponente muss in Isolation wie auch mit verschiedenen Konfigurationen und an unterschiedlichen Umgebungen getestet werden [12]. Wie in Kapitel 2 deutlich gemacht wurde, ist auch ein zusätzlicher Aufwand für Dokumentation und Zertifikation erforderlich. Für die Entwicklung von Komponenten bieten sich die evolutionären Prozessmodelle meistens besser an, da frühzeitiger als bei sequentiellen Modellen erste lauffähige Resultate entstehen. Wie später noch erläutert wird, kann es durchaus vorkommen, dass bei CBD Komponenten im eigenen Unternehmen erzeugt werden müssen. Dabei ist es für die CBD hilfreich während des Systemdesigns und der Integrationsphase auf Prototypen zurück greifen zu können. Ebenso kann es aufgrund der Time-to-Market-Anforderung auch für Lieferanten von Vorteil sein, evolutionär vorzugehen.

Komponentenbasierter Systementwicklungsprozess. Ein komplettes System aus bereits vorhandenen Komponenten zu konstruieren verspricht weniger Aufwand in der Implementierungsphase. Zusätzlicher Aufwand kommt aber für komponentenspezifische Aktivitäten, wie Suchen, Finden, Auswählen, Evaluieren und Testen hinzu [22]. Dennoch bleiben die grundsätzlichen Basisaktivitäten der Softwareentwicklung erhalten. Anforderungsanalyse und Spezifikation, Systemdesign, Implementierung, Test, Auslieferung und Wartung stellen die Aktivitäten des klassischen Wasserfallmodells dar. In Abb. 2 ist zu sehen, dass die meisten traditionellen Aktivitäten auch bei CBD weiterhin, wenn auch in modifizierter Form, vorhanden sind. Lediglich die Implementierungsphase scheint sich aufgelöst zu haben. Wie oben bereits beschrieben, tritt an diese Stelle eine neue Aktivität.

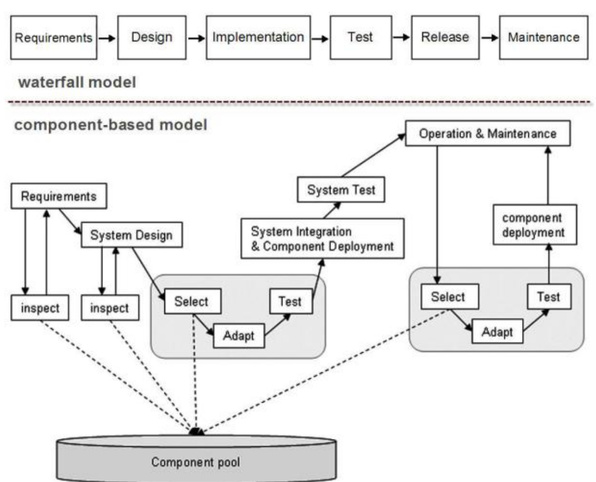


Abb. 2: komponentenbasierter Entwicklungsprozess vgl. [12]

Die Ausarbeitungen von Crnkovic [11][12] beleuchten die wichtigsten Unterschiede der einzelnen Phasen.

Anforderungsanalyse und Spezifikation: In dieser Phase ist es wichtig zu analysieren, ob die Anforderungen mit verfügbaren Komponenten realisierbar sind. Lassen sich keine entsprechenden Komponenten finden, muss die benötigte Komponente im eigenen Haus zur Entwicklung angestoßen werden oder die Anforderung auf zur Verfügung stehende Komponenten angepasst werden.

Systemdesign: Da Komponenten meist im Kontext von Komponentemodellen entwickelt werden, ist Interoperabilität zwischen Komponenten unterschiedlicher Technologie nur schwer zu erreichen. Dies führt zu einer Einengung von Architektur- und Systemdesignentscheidungen. Diese Phase wird bedeutend von den

Charakteristika der verfügbaren Komponenten und dem gewählten Komponentenmodell beeinflusst, auf welche der Systemarchitekt während der gesamten Phase Rücksicht nehmen muss.

Implementierung und Komponententest: Wie in Abb. 2 ersichtlich, folgt dem Design die Phase der Auswahl, Anpassung und Überprüfung der vorselektierten Komponenten. Funktionen und Spezifikationen der ausgewählten Komponenten müssen isoliert verifiziert und getestet werden. Danach werden Komponenten assembliert und die dabei entstandenen Baugruppen erneut auf korrekte Funktion getestet. Oftmals müssen Komponenten noch mit Adaptern ausgestattet werden, um mit der darunter liegenden Architektur kompatibel zu sein. Die Implementierungsphase besteht nur noch teilweise aus Programmieren. Code zu schreiben beschränkt sich im Idealfall auf die Erzeugung von „glue code“ (Code zur Verbindung von Teilen beim Assemblieren).

System Integration: Diese Phase wird bedeutend von der gewählten Komponententechnologie bestimmt, da nun die Komponenten in das Komponentenmodell integriert und auf Architekturverträglichkeit geprüft werden müssen. Da die Komponententechnologie die Architektur weitgehend standardisiert, können an dieser Stelle die Vorteile von Architekturmustern ausgeschöpft werden.

System Test: Die Systemtestmethoden unterscheiden sich nur wenig vom Standardvorgehen. Die Besonderheiten liegen in der Blackbox-Sichtbarkeit von Komponenten begründet. Die Fehlerzuordnung zu einer Blackbox-Komponente kann bei Auftreten eines Fehlers, ob der fehlenden Implementierungsdetails, erschwert sein. In solchen Fällen sind vertragliche Schnittstellen und die Möglichkeit beim Komponentenhersteller bezüglich technischer Details rückfragen zu können [22] das beste Gegenmittel.

Auslieferung und Wartung: Die Auslieferung eines komponentenbasierten Systems unterscheidet sich nicht von der Auslieferung anderer Systeme. Entscheidende Unterschiede gibt es im Wartungsprozess. Idealerweise sind einfach alte durch neue Komponenten zu ersetzen. Oder bei Funktionserweiterung zusätzliche Bauteile ins System zu integrieren. Der Prozess dafür ist wie zuvor schon: Suchen, Finden, Adaptieren, Testen und Integrieren.

Es soll nicht unerwähnt bleiben, dass das Finden der richtigen Komponenten nach wie vor eine große Herausforderung darstellt. Um dem gerecht zu werden sind ein reichhaltiger Komponentenmarkt [6] als auch unterstützende Tools erforderlich [13]. Das Lebenszyklusmodell muss noch reifen und vertrauenswürdige zertifizierte Komponenten sind rar. In der Literatur finden sich gescheiterte [17] wie auch erfolgreiche komponentenbasiert entwickelte Projekte. Sparling gibt uns in [26] Hinweise dazu, welche Aspekte zur erfolgreichen Entwicklung komponentenbasierter Systeme beitragen.

5. Zusammenfassung

CBSE hat als Subdisziplin des Software-Engineerings in den letzten Jahren an Aufmerksamkeit gewonnen. Dennoch ist die Definition für den Softwarekomponentenbegriff zu vielschichtig und müsste präzisiert werden. Die gezielte Forschung nach qualitätssichernden Methoden bestätigt, dass Qualität mit ein entscheidender Faktor für die Akzeptanz von Softwarekomponenten ist. Erst vertrauenswürdige Komponenten hoher Qualität werden in der Lage sein, die Hürde des „not-invented-here“-Syndroms zu nehmen. Die andiskutierten Methoden der Dokumentation, Qualitätsmodelle und Zertifizierung werden ihren Teil dazu beitragen. Die meisten Komponenten sind ohne Infrastruktur nicht lauffähig. Komponentenmodelle wie CORBA/CCM, EJB und .NET stellen für den Betrieb wichtige Dienste bereit und halten die Komponenten förmlich zusammen. Die Komponententechnologie kämpft jedoch nach wie vor mit der Problematik der Interoperabilität, wobei der Adapter als eine Lösung für diese Schwachstelle angesprochen wurde. Die Entwicklung komponentenbasierter Softwaresysteme erfordert ein speziell angepasstes Lebenszyklusmodell. Suchen, Auswählen und Adaptieren von verfügbaren Komponenten kommen als neue Aktivitäten im Prozessmodell hinzu. Das vorgestellte Lebenszyklusmodell ließ die Unterschiede der CBD zu herkömmlichen Prozessaktivitäten sichtbar werden.

Peter Wegner hat bereits 1984 demonstriert wie kapitalintensiv Softwaretechnologie ist. Anhand der in diesem Artikel vorgestellten heute verfügbaren Softwarekomponenten-Technologie wird deutlich, dass die Kapitalintensität noch weiter zugenommen hat. Komponenten hoher Qualität, teure Middleware und speziell abgestimmte gut implementierte Prozesse machen die Konstruktion und den effektiven Einsatz dieser Technologie erst möglich.

Referenzen

[1] Alvaro A., Santana de Almeida E., Romero de Lemos Meira S., Software Component Certification: A Survey, *euromicro*, 2005, pp. 106-113.

[2] Andreou A. S., Tziakouris M., A quality framework for developing and evaluating original software components. *Inf. Softw. Technol.* 49, 2 (Feb. 2007), pp. 122-141.

[3] Bachmann F. et al., Volume II: Technical Concepts of Component-Based Software Engineering, *Technical Report, Software Engineering Institute (SEI)*, May, 2000, pp. 65.

[4] Bertoa M. F., Vallecillo A., Quality attributes for cots components, *in Proceedings of the 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering,* QAOOSE 2002.

[5] Boegh J., Certifying Software Component Attributes, *IEEE Software*, 23, 3 (May 2006), pp. 74-81.

[6] Brereton P. et al., Software Components - Enabling a Mass Market, *In Proceedings of the 10th international Workshop on Software Technology and Engineering Practice*, IEEE Computer Society, Washington, DC, 2002.

[7] Broy M. et al., What characterizes a (software) component?, *Software–Concepts and Tools*, 19, 1(1998), pp. 49-56.

[8] Buchi, M., Weck, W., The Greybox Approach: When Blackbox Specifications Hide Too Much, *Technical Report: UMI Order Number: TUCS-TR-297*, Turku Centre for Computer Science, 1999.

[9] Chiang C. C., The use of adapters to support interoperability of components for reusability, *Information Software Technology*, 3 (2003), pp. 149-156.

[10] Correira J. M., Biscotti F., Market Share: AIM and Portal Software, Worldwide, 2005. *Gartner market research report*, Gartner, June 2006.

[11] Crnkovic I. et al., Component-Based Development Process and Component Lifecycle, *icsea*, 2006, p. 44.

[12] Crnkovic I., Chaudron M., Larsson S., Component-Based Development Process and Component Lifecycle, *27th International Conference Information Technology Interfaces* (ITI), IEEE, Cavtat, Croatia, 2005.

[13] Crnkovic I., Component-based Software Engineering - New Challenges in Software Development, *Software Focus*, December, 2001, John Wiley & Sons.

[14] Councill W. T., Heineman G. T., *Component-Based Software Engineering,* Addison-Wesley, 2001. p. xxii.

[15] DeMichiel L, Keith M., *JSR 220: Enterprise JavaBeans, Version 3.0, EJB 3.0 Simplified API*, SUN Microsystems, Dez. 2005.

[16] Emmerich W., Aoyama M., Sventek, J, The impact of research on middleware technology, *SIGSOFT Software Eng.. Notes* 32, 1 (Jan. 2007), pp. 21-46.

[17] Garlan, D., Allen, R., Ockerbloom, J, Architectural Mismatch: Why Reuse Is So Hard. *IEEE Softw.* 12, 6 (Nov. 1995), pp. 17-26.

[18] McIlroy M. D., Mass Produced Software Components, *NATO Software Engineering Conference Report*, Garmisch, Germany, October, 1968, pp. 79-85.

[19] Meyer B., On To Components, *IEEE Computer*, 32, 1 (Jan. 1999), pp. 139-143.

[20] Meyer B., The Grand Challenge of Trusted Components, *The IEEE Proc. of 25th International Conference on Software Engineering (ICSE),* USA, 2003, pp. 660–667.

[21] Mili, H., Mili, F., Mili, A., Reusing Software: Issues and Research Directions, *IEEE Trans. Softw. Eng.* 21, 6 (Jun. 1995), pp. 528-562.

[22] Morisio M. et al., Investigating and Improving a COTS-Based Software Development Process, *ICSE* 2000. Limmerick, Ireland.

[23] Sametinger J., *Software Engineering with Reusable Components*, Springer-Verlag, Berlin, 1997.

[24] Sarakatsanis A., Dinner for ONE?, *Java Magazin*, Software & Support Verlag, Frankfurt, 11.2001. pp. 60-65.

[25] Schneider J. G., Han J., Components — the Past, the Present, and the Future, *Proc. of 9th Int. Workshop on Component-Oriented Programming*, Oslo, Jun.2004, pp. 08.

[26] Sparling M., Lessons Learned Through Six Years of Component-Based Development, *Communications of the ACM*, 43, 10 (Oct. 2000), pp. 47-53.

[27] Stark T., *Java EE 5*, Pearson Studium, München, 2007.

[28] Szyperski C., *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, Essex, 1998.

[29] Taulavuori A., Niemelä E., Kallio P., Component documentation - a key issue in software product lines, *Inf. Softw. Technol.*, 46, 8 (2004), pp. 535-546.

[30] Wang N., Schmidt C. D., O´Ryan C., in Councill W. T., Heineman G. T., *Component-Based Software Engineering*, Addison-Wesley, 2001. pp. 557-571

[31] Wegner P., Capital-Intensive Software Technology. *IEEE Software*, 1, 3 (Jul. 1984), pp. 7-45.

[32] Wohlin, C., Runeson, P., Certification of Software Components. *IEEE Trans. Softw. Eng.* 20, 6 (Jun. 1994), pp. 494-499.

Faceted Classification

Christoph Kofler
0560345
christoph.kofler@edu.uni-klu.ac.at

ABSTRACT

In der heutigen Zeit der (komponentenbasierten) Softwareentwicklung wird das bereits sehr wichtige Thema „Software Reuse“ vermehrt in eine zentrale Position gestellt. Jedoch gibt es ein großes Problem, das den Reuse-Prozess erschwert: die Klassifizierung und Auffindung von bereits vorhandenen Komponenten.
Der vorliegende Artikel befasst sich nun mit der Klassifizierung solcher Software-Komponenten, um sie einzuordnen und Software-Entwicklern die Möglichkeit zu bieten, diese zu finden und bei Bedarf zu adaptieren. Ferner baut der Aufsatz auf den Grundprinzipien der Software-Klassifizierung auf. Dabei werden alte, grundlegende Überlegungen zu neuen, aufbauenden Ideen und Lösungen in Relation gesetzt.
Speziell wird ein Schema behandelt, welches davon ausgeht, dass Software-Collections sehr groß sind, ständig anwachsen und viele ähnliche Software-Komponenten beinhalten. Innerhalb dieses „Faceted Classification Scheme“ werden die Klassifizierung, ihr Umfeld und ein Library-System, welches auf diesem Schema aufbaut, diskutiert. Darüber hinaus werden moderne Ansätze wie Behavior Sampling und objektorientierte, feature-orientierte und semantik-basierte Klassifizierung behandelt, die das Thema weiterführen.

1. EINLEITUNG UND MOTIVATION

„Reuse is the use of previously acquired concepts and objects in a new situation“ [11]. Diese Definition von Software Reuse wurde von Prieto-Diaz und Freeman in [11] aufgestellt und zeigt die Mächtigkeit der Wiederverwendung bestehender Software-Komponenten: diese können zur Lösung neuer (Teil-)Probleme eingesetzt werden, ohne die benötigte Funktionalität erneut zu implementieren. Software Reuse bringt viele Vorteile wie Produktivitäts- und Qualitätssteigerung und leichtere Erweiterbarkeit von Systemen mit sich. Jedoch kommen diese Vorteile lediglich bei spezieller Organisation der Software-Bibliothek, welche die bereits existierenden Software-Komponenten beinhaltet, zu tragen.

Diese Organisation ist ein herausforderndes Problem, das in den letzten Jahrzehnten in zahlreichen Artikeln und Publikationen diskutiert wurde. Das Hauptproblem stellt dabei die Klassifizierung, also das wohldefinierte Einordnen existierender Software-Komponenten in die Software-Collection, dar. Aufgrund dieses Klassifizierungs-Problems wird die Wiederauffindung von Komponenten erschwert.

Prieto-Diaz und Freeman haben in [11] ein Reuse-Modell entwickelt, welches die oben beschriebenen Probleme erfasst und zu lösen versucht. Aus diesem Modell wurde anschließend ein Klassifikations-Schema, das „Faceted Classification Scheme“, definiert. Weiters realisierten Prieto-Diaz und Freeman in [11] ein konkretes Library-System, welches auf dem facetten-orientierten Klassifizierungs-Schema aufbaut.

Von der Arbeit von Prieto-Diaz und Freeman motiviert, entstanden im Laufe der Zeit weitere Schemata zur Klassifizierung von Software-Komponenten, die auf dem „Faceted Classification Scheme“ aufbauen.

Dieser Artikel diskutiert nun genauer das facetten-orientierte Klassifizierungs-Schema von Prieto-Diaz und Freeman aus [11] und setzt dieses zu aufbauenden Ansätzen und modernen Technologien in Relation. Die Arbeit ist wie folgt aufgebaut:

In Kapitel 2 wird beschrieben, warum und wie Software Reuse durchgeführt werden soll.

In Kapitel 3 wird aufgezeigt, dass die Wiederverwendung von Software-Komponenten nicht nur Vorteile, sondern auch bestimmte Nachteile mit sich bringt. Aufgrund der Problembeschreibung wird das Reuse-Modell motiviert und erörtert und gleichzeitig auf das vierte Kapitel übergeleitet.

In Kapitel 4 wird das „Faceted Classification Scheme“ diskutiert. Dabei wird näher auf die Herleitung der Facetten, die für die Beschreibung der Software-Komponenten in diesem Schema verantwortlich sind, auf einen Thesaurus und ein Distanz-Modell, eingegangen. Darüber hinaus wird das Library-System von Prieto-Diaz und Freeman in [11], das auf dem facetten-orientierten Klassifizierungs-Schema aufbaut, beleuchtet.

In Kapitel 5 werden anschließend Ansätze und moderne Technologien beschrieben, die auf der Arbeit von Prieto-Diaz und Freeman aufbauen. Dabei werden verschiedene Klassifizierungs-Schemata beschrieben und zu dem facetten-orientierten Ansatz in Relation gesetzt. Ebenso wird Behavior Sampling, eine weitere Wiederauffindungs-Methode für Software-Komponenten, diskutiert.

2. SOFTWARE REUSE

Software Reuse ist die Wiederverwendung von bereits existierenden Software-Komponenten bzw. Konzepten in neuen Situationen, Anwendungsproblemen und Systemen [11].

Warum und wie diese Wiederverwendung durchgeführt werden soll, wird im Folgenden erklärt und diskutiert.

2.1 Warum Software Reuse?

In vielen Disziplinen ist die Wiederverwendung von Wissen und bestehenden Komponenten eine Selbstverständlichkeit – man denke zum Beispiel an die Produktion eines Autos. In der Informatik ist dies jedoch anders: mit jedem Software-Projekt werden bestehende Ideen und Realisierungen neu umgesetzt [2, p. 1]. Jedoch bringt Software Reuse viele Vorteile mit sich [2, p. 1ff]:

- Produktivität
  Eine Software-Komponente soll dann wiederverwendet werden, wenn der Aufwand im Vergleich zu einer Neuentwicklung geringer wird. Ist dies der Fall, steigt damit gleichzeitig die Produktivität und Effizienz.
- Qualität und Zuverlässigkeit
  Durch die (mehrmalige) Wiederverwendung einer Software-Komponente werden evtl. existierende Fehler in dieser beseitigt. Dadurch steigt parallel die Qualität und Zuverlässigkeit des zu entwickelnden Gesamtsystems.
- Erweiterbarkeit
  Ein System, das aus wiederverwendeten Komponenten besteht, ist leichter zu erweitern. Außerdem entsteht dabei ein ständiger Lernprozess für den Entwickler (durch Techniken und Ansichten, die der ursprüngliche Autor verwendet hat).

Software Reuse ist nur bei entsprechender Organisation des Reuse-Prozesses, wie in [7] beschrieben, sinnvoll.

2.2 Reuse-Prozess

Der Software Reuse-Prozess ist ein Lebenszyklus, der auf einer Bibliothek aufbaut und aus folgenden Phasen besteht [12, p. 46]:

1. Einordnen („Classifying“) von Software-Komponenten, die zur Wiederverwendung bereitgestellt werden.
2. Späteres Wiederauffinden („Retrieval“) dieser Komponenten von anderen Entwicklern.
3. Entwicklung neuer Software aufgrund der gefundenen Software-Komponenten.

Dieser Prozess soll für alle Beteiligten so einfach als möglich gehalten werden, da andernfalls auf den Reuse von existierenden Komponenten verzichtet und die benötigte Funktionalität selbst entwickelt wird [12, p. 46].

3. PROBLEMBESCHREIBUNG

Dass Software Reuse nicht nur Vorteile, sondern auch Kritikpunkte mit sich bringt, wird durch den Reuse-Prozess ersichtlich. Alle Vorzüge, die in Kapitel 2.1 beschrieben wurden treten also nur in Kraft, wenn die Hauptprobleme des Reuse-Prozesses so gut als möglich beseitigt werden [11, 12, p. 46]. Diese wären [11]:

- Klassifizierung (Classifying)
  Eine wohldefinierte Klassifizierung einer Komponente und die dadurch entstehende Ordnung und Organisation in Software-Collections sind essenzielle Bestandteile der Wiederverwendung von Software.
- Wiederauffindung (Retrieval)
  Dieses Problem steht mit dem Klassifizierungs-Problem in Verbindung. Werden Software-Komponenten nicht geeignet klassifiziert und beschrieben, stellt auch die Wiederauffindung dieser ein herausforderndes Problem dar.

Werden Software-Bibliotheken also nicht geeignet organisiert, steigt dadurch die Wahrscheinlichkeit, dass ein Entwickler die passende Komponente nicht finden kann und sie somit selbst entwickelt.

Natürlich wird bei Software Reuse nicht nur der Software-Collection, in der Komponenten organisiert sind, eine zentrale Problem-Rolle zugesprochen. Auch die Entwickler, die Benutzer dieser Bibliotheken, bilden ein (Teil-)Problem. Diese müssen folgenden Prozess durchlaufen, um eine bestehende Software-Komponente in neuen Systemen einzusetzen: (1) die richtige (!) Komponente finden, (2) diese verstehen und (3) die Komponente, falls notwendig, so adaptieren, dass sie die Funktionalität in der neuen Umgebung bietet. Diese Aussage impliziert nun, dass Attribute wie Größe, Komplexität und Dokumentation, die ausschließlich von Entwicklern abhängen, ein Problem der Software-Wiederverwendung darstellen [11].

Aus diesem Grund haben Prieto-Diaz und Freeman in [11] ein Reuse-Modell hergeleitet, das neben den oben beschriebenen Problemen auf die Annahmen, dass Collections sehr groß sind, ständig anwachsen und viele ähnliche Software-Komponenten beinhalten, aufbaut [11].

Weiters wird angenommen, dass aufgrund der vielen ähnlichen Komponenten bei der Suche nach diesen keine eindeutige Komponente retourniert werden kann, und damit die Adaptierung von Komponenten keine Ausnahme darstellt [11].

Der folgende Algorithmus beschreibt das Reuse-Modell treffend [11]:

```
begin
  search library
  if identical match then terminate
  else
    collect similar components
    for each component
      compute degree of match
    end
    rank and select best
    modify component
  fi
end
```

Listing 1: Reuse-Modell-Algorithmus (Quelle: [11])

Wie aus dem Algorithmus aus Listing 1 zu entnehmen ist, wird zuerst die Software-Library nach der gewünschten Komponente durchsucht. Wird eine exakte Software-Komponente gefunden, terminiert der Algorithmus. Ist dies nicht der Fall, werden ähnliche Komponenten aufgelistet, und mit Hilfe eines Evaluierungs-Mechanismus, der ebenso Bestandteil des Modells ist, bewertet. Anschließend wird die beste Komponente, also diese, die am wenigsten Adaptierungsaufwand erfordert, ausgewählt [11].

4. FACETED CLASSIFICATION

Das Reuse-Modell stellt einen Lösungsansatz (= Modell) dar, der von Prieto-Diaz und Freeman in [11] in eine konkrete Lösung, das „Faceted Classification Scheme“, übergeleitet wurde und die Umsetzung aller Annahmen und Anforderungen, die im Modell entstanden sind, beinhaltet.

Im folgenden Kapitel wird nun auf die Herleitung dieses Schemas und eine konkrete Realisierung näher eingegangen.

4.1 Komponenten-Beschreibung

Bei der facetten-orientierten Klassifikation wird die Beschreibung einer Software-Komponente in elementare Klassen zerlegt und diese anschließend mit Facetten beschrieben. Diese Beschreibung soll kurz und bündig ausfallen. Die Klassifizierung wird anschließend durch die Gruppierung der Facetten gewährleistet. Diese Art von Klassifizierung ist leicht erweiterbar, flexibel, und für sehr große, ständig anwachsende Software-Collections ausgezeichnet geeignet [8, 11].

Prieto-Diaz und Freeman bauen in [11] auf den Ansatz auf, dass Software-Komponenten mit den folgenden beiden Beschreibungen charakterisiert werden können [11]:

- Funktionalität (Functionality)
- Umgebung (Environment)

Durch Analysieren dieser Beschreibungen wurden konkrete Facetten hergeleitet, die für die Beschreibung von Software-Komponenten im facetten-orientierten Schema dienen.

Funktionalität

Existieren beispielsweise zwei Software-Komponenten, welche die gleiche Funktionalität realisieren, bedeutet dies nicht zwingend, dass die beiden Komponenten gleich sind. Man denke zum Beispiel an die Implementierung einer Suchfunktion in einem Compiler und in einem Datenbanksystem. Die Suche im Compiler operiert evtl. auf einer Symbol-Tabelle, wobei im Datenbanksystem ein B-Baum als Datenstruktur herangezogen werden könnte [11].

Aus diesem Grund wurden neben der Funktion („function“), die eine Komponente ausführt, noch zwei weitere Facetten definiert, um „Funktionalität“ zu beschreiben: Objekt („object“) und Medium („medium“). Die Facetten werden in Form des Tupels <function, object, medium> dargestellt und haben folgende Bedeutung [5, 11]:

- Function
  Name der Funktion, welche die Komponente ausführt (z.B. „search“, „add“).
- Object
  Objekt, das von der Komponente manipuliert und zur Ausführung der Funktionalität benötigt wird (z.B. „arguments“, „root“).
- Medium
  Medium, das verwendet wird, um die Funktionalität auszuführen. Stellt eine größere Datenstruktur dar, welche das manipulierende Objekt beinhaltet (z.B. „B-tree“, „array“).

Umgebung

Diese Beschreibung charakterisiert den Kontext, für den die Software-Komponente entwickelt wurde [5]. Für eine Komponente existieren zwei Umgebungen [11]:

- Interne Umgebung (Internal Environment)
  Die interne Umgebung bezieht sich auf das Umfeld, in dem die Software-Komponente ausgeführt wird. Ändert sich diese Umgebung, hat dies keine relevante Auswirkung auf die Funktionalität der betroffenen Komponente, da diese üblicherweise unverändert bleibt.
- Externe Umgebung (External Environment)
  Die externe Umgebung stellt das Umfeld dar, in dem die Komponente angewandt wird. Ändert sich diese Umgebung, muss das Design bzw. die Spezifikation der Software-Komponente an die neue Umgebung angepasst werden. Dies impliziert, dass sich auch die Funktionalität ändert.

Aus diesem Grund beziehen sich Prieto-Diaz und Freeman in [11] auf die externe Umgebung, da diese mehr Einfluss auf die Wiederverwendbarkeit einer Software-Komponente hat.

Wird nun diese externe Umgebung analysiert, entstehen folgende Facetten, die „Umgebung“ beschreiben: System-Typ („system type“), Functional Area („functional area“) und Setting („setting“). Die Facetten werden in Form des Tupels <system type, functional area, setting> dargestellt und haben, geordnet nach absteigender Reusability-Relevanz, folgende Bedeutung [5, 11]:

- System Type
  (Sub-)Systemtyp, für den die Komponente entwickelt wurde (applikationsunabhängig; z.B. „interpreter“, „DB management“).
- Functional Area
  Applikationsabhängige Aktivitäten, die von der Software-Komponente durchgeführt werden (z.B. „batch job control“, „CAD“).
- Setting
  Bereich, in der die Software-Komponente ausgeführt wird (z.B. „auto repair“, „catalog sales“).

Durch das Sechs-Tupel <function, object, medium, system type, functional area, setting> wird also im facetten-orientierten Klassifikations-Schema eine Software-Komponente passend klassifiziert.

4.2 Thesaurus

Die Beschreibung von Software-Komponenten im „Faceted Classification Scheme“ ist nicht problemfrei, da bei der Beschreibung einer Komponente mehrere Synonyme für die selbige entstehen können. Man denke zum Beispiel an die Beschreibung der Funktionalität einer Software-Komponente: dabei könnte diese in Form von <move, words, file> oder <transfer, names, file> beschrieben werden. Dabei entstehen mehrere Beschreibungen für die selbe Komponente [11].

Um diese Probleme zu beseitigen, ist eine Vokabular-Kontrolle, der Thesaurus, entstanden. Dieser gruppiert alle Synonyme einer Facette unter einem eindeutigen, aussagekräftigen Begriff. Dabei wird der Begriff, der die Facette für die betroffene Software-Komponente am besten beschreibt, verwendet (z.B. add: increment/total/sum) [6, 11].

Der Thesaurus kann auch die Anzahl der Suchergebnisse beeinflussen: einerseits können viele verschiedene Begriffe zu wenigen Gruppen zusammengefasst werden (mehr Ergebnisse), andererseits ist es auch möglich, diese Gruppen zu spezialisieren (weniger Ergebnisse) [11].

4.3 Distanz-Modell

Auch das Distanz-Modell stellt ein wichtiges Feature im facetten-orientierten Klassifikations-Schema dar. Dabei handelt es sich um die „semantische Distanz“ zwischen zwei Begriffen einer Facette.

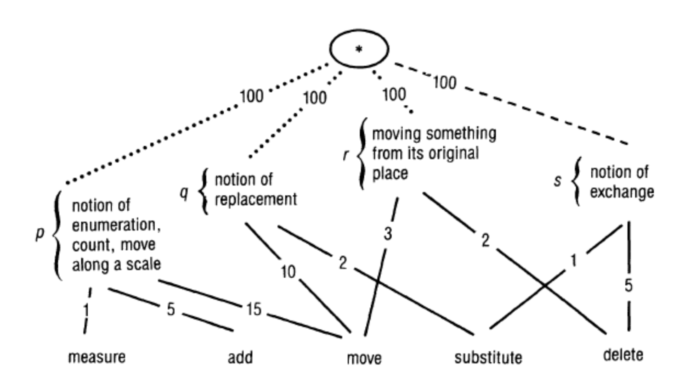


Abbildung 1: Distanz-Modell (Quelle: [11])

Wie in Abbildung 1 zu sehen ist, werden die Begriffe, die eine Facette beschreiben, in Form von Blättern in einem azyklischen, gerichteten Graph gespeichert. Knoten des Graphen stellen dabei Verbindungen zwischen ähnlichen Begriffen dar. Die Bewertung der Kanten ist benutzerspezifisch: je größer die Relation eines Begriffes zu einem Knoten ist, um so niedriger wird diese Kante bewertet [11].

Das Beispiel in Abbildung 1 zeigt das Distanz-Modell der „function“-Facette. Dabei wird ersichtlich, dass zum Beispiel „measure“ näher bei „add“ ($1 + 5 = 6$) als bei „move“ ($1 + 15 = 16$) liegt [11].

Dieses Modell wird bei der Wiederauffindung von Software-Komponenten benötigt: falls die Software-Collection aufgrund bestimmter Suchbegriffe keine Ergebnisse liefert, wird das Distanz-Modell herangezogen, um die gleiche Suche mit den ähnlichsten Begriffen dieser Facette zu wiederholen [11].

Neben den erklärten Vorteilen, die das Modell mit sich bringt, existiert jedoch auch ein Nachteil: das Aufbauen des azyklischen, gerichteten Graphen ist zeitintensiv [11].

4.4 Library-System

Das soeben beschriebene Klassifikations-Schema wurde zusammen mit dem Thesaurus und dem Distanz-Modell zusätzlich zu einem Mechanismus zum Evaluieren und Ordnen von Komponenten-Suchergebnissen in Form eines Library-Systems von Prieto-Diaz und Freeman in [11] realisiert und umgesetzt.

Die so entstandene Software-Collection bietet damit die im Reuse-Modell definierte Funktionalität. Um das „Faceted Classification Scheme“ wie beschrieben umzusetzen, teilt sich das Library-System in zwei Teile [11]:

- Query-System
- Evaluierungs-Mechanismus

Wie bereits bei der Komponenten-Beschreibung in Kapitel 4.1 ersichtlich wurde, besteht die Beschreibung einer Software-Komponente im facetten-orientierten Klassifikations-Schema aus einem Sechs-Tupel. Das Query-System baut nun auf dieser Beschreibung auf und setzt sich aus folgenden Komponenten zusammen [11]:

- Query-Generierung
  Bei der Query-Generierung werden die Facetten mit Werten befüllt. Spielt der Wert einer Facette keine Rolle, ist es möglich, diese Facette mit einer „Wildcard“ in Form eines Sternes zu beschreiben. Liefert eine Query kein zufriedenstellendes Suchergebnis (z.B. kein Ergebnis oder zu viele gefundene Komponenten), kann diese modifiziert werden.
- Query-Modifizierung
  Soll eine Query allgemeiner werden, d. h. mehr Suchergebnisse liefern, können einzelne Facetten mit Hilfe von Wildcards beschrieben werden. Wird jedoch das Gegenteil benötigt, d. h. eine Query lieferte zu viele Suchergebnisse und soll nun spezieller werden, müssen Wildcards durch konkrete Werte ersetzt werden.
- Query-Erweiterung
  Um eine Query-Erweiterung zu gewährleisten, stützen sich Prieto-Diaz und Freeman in [11] auf das Distanz-Modell. Liefert eine Query kein (zufriedenstellendes) Komponenten-Suchergebnis, wird der azyklische, gerichtete Graph herangezogen um den ähnlichsten Begriff der aktuellen Facetten-Beschreibung zu ermitteln.

Wurde eine Query erfolgreich ausgeführt, gehen Prieto-Diaz und Freeman in [11], wie bereits erwähnt, davon aus, dass das Library-System sehr groß ist, ständig anwächst und sehr viele ähnliche Komponenten beinhaltet. Aus diesem Grund ist die Wahrscheinlichkeit eines exakten Suchtreffers gering.

Um aus der erhaltenen Liste von Suchergebnissen die passendste Komponente auszuwählen, wird ein Evaluierungs-Mechanismus benötigt. Dieser baut auf den folgenden Attributen, die eine Software-Komponente im Library-System zusätzlich beschreiben, auf [11]:

- Größe
- Simplizität
- Dokumentation
- Programmiersprache, in der die Komponente implementiert wurde

Die Software-Komponente, die diese vier Attribute am besten erfüllt, sollte für den Reuse ausgewählt werden.

5. AUFBAUENDE ANSÄTZE UND MODERNE TECHNOLOGIEN

Im bisherigen Teil des Artikels wurde die Klassifizierung von Software-Komponenten, um diese wiederzuverwenden, durch die Erklärung von Software Reuse und dessen Problembeschreibung motiviert. Anschließend wurde das facetten-orientierte Klassifikations-Schema, das von Prieto-Diaz und Freeman in [11] hergeleitet wurde, näher beleuchtet.

Nun werden Ansätze und moderne Technologien, die auf der Arbeit von Prieto-Diaz und Freeman aufbauen, diskutiert und zu dieser in Relation gesetzt.

5.1 Objekt-orientierte Klassifizierung

In der heutigen Zeit der Softwareentwicklung ist ein objekt-orientierter Software-Entwicklungsprozess nicht mehr wegzudenken. Aus diesem Grund befassen sich Karlsson et al. in [4] ausführlich mit dem Thema des objekt-orientierten Software Reuse.

In diesem Kapitel werden nun die Unterschiede der Wiederverwendung und der Klassifizierung im Vergleich zur Arbeit von Prieto-Diaz und Freeman in [11] diskutiert.

Wird bei der Lösung von Prieto-Diaz und Freeman von „traditioneller“ Programmierung, d. h. von Prozeduren, Funktionen und Blöcken von Source-Code ausgegangen, steht bei der Weiterentwicklung von Karlsson et al. in [4] die Beschreibung eines Objekts – die Klasse – im Mittelpunkt [4].

Daher wird bei der objekt-orientierten Wiederverwendung von Software-Komponenten ein konkreter Vorteil genutzt: die Vererbung. Eine Subklasse erbt die Funktionalität ihrer Reuse-Superklasse. Dabei entstehen folgende Möglichkeiten der Wiederverwendung [4]:

- Reuse einer bestehenden Klasse, ohne neue Funktionalität in einer Subklasse zu implementieren. Dies ist natürlich die komfortabelste, jedoch auch unwahrscheinlichste Variante.
- Reuse der bestehenden Funktionalität durch eine Superklasse und Hinzufügen bzw. Überschreiben von Methoden in einer Subklasse, um die neue Funktionalität zu gewährleisten.
- Reuse durch Modifikation der Funktionalität einer Komponente. Dabei wird die Idee der bestehenden Software-Komponente übernommen und auf die neue Umgebung angepasst. Dies ist die wahrscheinlich aufwändigste Variante, da Source-Code für die neue Umgebung wiederum neu implementiert werden muss.

Da, wie bereits erwähnt, bei diesem objekt-orientierten Ansatz die Klasse im Mittelpunkt steht, müssen auch die Facetten von Prieto-Diaz und Freeman aus [11] modifiziert werden [4].

Karlsson et al. definieren in dem Reuse-System REBOOT[1], das in [4] eingeführt wurde und auf die Klassifizierung von objekt-orientieren Software-Komponenten setzt, vier Facetten, wobei sich davon drei auf die Funktionalitäts-Facetten, und lediglich eine auf die Umgebungs-Facetten aus dem Ansatz von Prieto-Diaz und Freeman in [11] beziehen. Die vier Facetten sind [4]:

- Abstraction
  Diese Facette wurde definiert, da objekt-orientierte Komponenten für gewöhnlich eine Abstraktion eines Objektes der realen Welt darstellen. Die Objekte werden mit einem speziellen Vokabular (Name), wie z.B. Stack, Queue etc., beschrieben, welches auch für die Klassifizierung der Software-Komponente herangezogen wird.
- Operations
  Objekt-orientierte Komponenten bieten Methoden an, welche die Funktionalität der Software-Komponente bzw. des Objektes charakterisieren. Die Operations-Facette soll nun eine Liste von Operationen, die von einer Komponente angeboten werden, beinhalten (z.B. push, pop). Falls es sich bei der Software-Komponente um eine einzige Klasse handelt, sollen lt. Karlsson et al. in [4] nur die öffentlichen (public) Methoden aufgelistet werden.
- OperatesOn
  Die OperatesOn-Facette beinhaltet die Objekte, die von der Software-Komponente bzw. von der Klasse verwaltet werden (u. a. In- und Output-Argumente).
- Dependencies
  Die letzte Facette ersetzt die drei Umgebungs-Facetten aus der Arbeit von Prieto-Diaz und Freeman in [11]. Die Dependencies-Facette beinhaltet alle Abhängigkeiten, welche die Verwendung der Software-Komponente einschränken könnte. Karlsson et al. machen in [4] jedoch keine konkreten Vorschläge, welche genauen Werte diese Facette annehmen sollte. Mögliche Beispiele wären: Design-Methodologie, Programmiersprache und Hardware.

Ähnlich wie bei der Arbeit von Prieto-Diaz und Freeman in [11] existiert auch im REBOOT-System von Karlsson et al. in [4] ein Modell, das dem Distanz-Modell nachempfunden ist und auf die „semantische Distanz“ zwischen verschiedenen Facetten-Beschreibungen abzielt. Dieses wird für den

[1]REBOOT – „REuse Based on Object-Oriented Techniques“.

„Search and Retrieval“-Mechanismus verwendet, der folgende besondere Funktionalität anbietet [4]:

- Query erweitern, dass ähnliche, jedoch nicht identische Komponenten in den Suchergebnissen aufscheinen.
- Finden einer bestimmten Software-Komponente, obwohl diese evtl. falsch klassifiziert wurde (durch sog. „Semantic Matching“).

Ein weiterer objekt-orientierter Klassifizierungsansatz wird in [6] diskutiert. Dabei werden die Funktionalitäts-Facetten von Prieto-Diaz und Freeman aus [11] nicht verändert, ihnen jedoch teilweise eine andere Bedeutung zugesprochen.

5.2 Feature-orientierte Klassifizierung

Die Lösungen von Prieto-Diaz und Freeman in [11] und Karlsson et al. in [4], die für die Beschreibung von Software-Komponenten auf Facetten aufbauen, haben einen Nachteil: die Anzahl der Facetten ist von der Implementierung des Library-Systems abhängig (z.B. sechs Facetten bei dem Ansatz von Prieto-Diaz und Freeman in [11] und vier Facetten bei der Arbeit von Karlsson et al. in [4]). Dadurch wird auch das Beschreiben von Komponenten, die mehrere Funktionen anbieten, schwierig [1].

Aus diesem Grund geht Börstler in [1] einem feature-orientierten Ansatz nach, der die Probleme der facetten-orientierten Klassifizierung beseitigen soll. Im Zuge dieses Ansatzes realisiert Börstler ein feature-orientiertes Klassifikations-Schema mit dem Namen FOCS[2].

Bei der feature-orientierten Klassifizierung wird eine Software-Komponente durch eine Menge von Eigenschaften (Features) beschrieben [1].

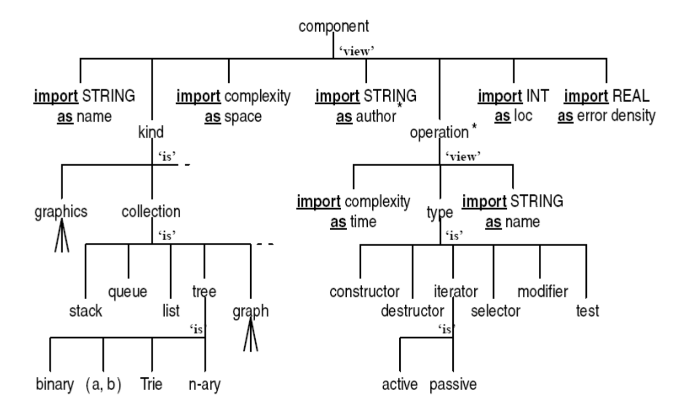


Abbildung 2: Feature-orientiertes Klassifizierungs-Schema (Quelle: [1])

Wie in Abbildung 2 ersichtlich ist, werden die Eigenschaften in Form eines feature-orientierten Klassifikations-Schemas organisiert und durch ständige Weiterzerlegung klassifiziert. Durch diese Weiterzerlegung entsteht ein Baum, bei dem ein Feature durch einen Pfad beschrieben werden kann [1].

[2]FOCS – „Feature-Oriented Classification System“.

In dem Schema von Börstler in [1] werden zwei Relationen für die Weiterzerlegung der Eigenschaften in diesem Baum unterstützt [1]:

- View-Refinement
  Diese Art von Refinement zerlegt ein Feature in klar erkennbare und unterscheidbare Aspekte. In dem Schema aus Abbildung 2 kann eine Komponente durch die folgenden Eigenschaften klassifiziert werden: Name (name), Art der Komponente (kind), Platz-Anforderungen (space), Autoren (author*), Operationen (operations*), Größe in Lines of Code (loc) und Fehlerdichte (error density).
- Is-Refinement
  Diese Weiterzerlegung wird für die Spezialisierung von Eigenschaften benötigt. Eine Komponente kann daher durch genau eine Spezialisierung eines Features klassifiziert werden. In dem Klassifikations-Schema aus Abbildung 2 wird beispielsweise eine „collection“ in „stack“, „queue“, „list“ etc. spezialisiert.

Sind für ein Feature mehrere Ausprägungen relevant, wird dies im Schema von Börstler durch einen Stern symbolisiert. Dies bedeutet im konkreten Schema aus Abbildung 2, dass die Eigenschaften „Author“ und „Operations“ mehrere Werte annehmen können [1]. Dies stellt eine deutliche Verbesserung der Ansätze aus [11] und [4] dar.

Folgendes Beispiel soll die Thematik der feature-orientierten Klassifizierung verdeutlichen [1]:

```
(
  name→ExtendedExample,
  author→ck,
  kind→collection→stack,
  space→constant,
  operation→(type→constructor, name→push,
             time→constant),
  operation→(type→destructor, name→pop,
             time→constant),
  loc→100,
  error density→0.01
)
```

Auch die Suche in der Software-Library wird mit der Unterstützung von Eigenschaften durchgeführt: ein Benutzer beschreibt die Query, die abgesetzt werden soll, in Form von Eigenschaften einer Software-Komponente. Dabei ist zusätzlich zu erwähnen, dass, sofern nach einer Generalisierung gesucht wird, ebenso die Spezialisierungen berücksichtigt werden (z.B. die Suche nach kind→collection→tree würde auch kind→collection→tree→binary berücksichtigen) [1].

Weiters wurde im Ansatz von Börstler in [1] ein Mechanismus eingeführt, der beispielsweise für die Suche nach kind→collection→tree→binary auch die Komponenten finden würde, die mit kind→collection→tree→n-ary klassifiziert wurden. Jedoch wird dafür ein Faktor berücksichtigt, der aussagt, dass dies kein exakter Treffer ist. Diese Abhängigkeiten unter den einzelnen Spezialisierungen werden ebenso im Klassifizierungs-Schema durch spezielle Relationen definiert [1].

5.3 Semantik-basierte Klassifizierung

Der semantik-basierte Klassifizierungs-Ansatz, der von Yao und Etzkorn in [13] verfolgt wird, bezieht sich auf das World Wide Web als Software-Bibliothek, die wiederverwendbare Software-Komponenten beinhaltet [13].

Dieser Ansatz baut auf Semantic Web [3], DAML+OIL[3] (später durch OWL[4] abgelöst), RDF[5], WSDL[6] und UDDI[7] auf. RDF besteht aus einem „object-attribute-value"-Modell, wobei „object" eine Ressource, „attribute" eine Eigenschaft und „value" einen freien Text darstellen. Durch dieses Tripel kann Information ohne Verlust an Sinnhaftigkeit zwischen mehreren Applikationen ausgetauscht werden. Die Ontologie-Sprache DAML+OIL wurde ebenso speziell für das Semantic Web realisiert und erweitert RDF. WSDL ist eine auf XML aufbauende Sprache, die Web Services beschreibt. UDDI ist ein Verzeichnis, bei dem Web Services registriert werden, um diese wiederzufinden [13].

Ausgehend von diesen Technologien wurde in [13] ein Ansatz für semantik-basierte Klassifizierung definiert, welcher aus drei Abschnitten besteht und durch Abbildung 3 zum Ausdruck gebracht wird.

In Abschnitt 1 wird ein intelligentes, auf natürlicher Sprache aufbauendes User Interface definiert. Durch dieses Interface ist ein Benutzer in der Lage, Queries durch natürliche Sprache auszudrücken, welche durch einen konzeptionellen Graphen (CG) repräsentiert und über Semantic Web zum Ausdruck gebracht wird. Weiters wird die natürliche Sprache in RDF bzw. WSDL umgewandelt. Um dies zu ermöglichen, wird auf DAML+OIL zurückgegriffen [13].

Im zweiten Abschnitt des Ansatzes wird ein Analyse-Tool definiert, welches bestehende Software-Komponenten durch eine semantik-basierte Beschreibung charakterisiert. Ebenso wie in Abschnitt 1 wird hier ein konzeptioneller Graph eingesetzt, um die Beschreibung zu repräsentieren und über Semantic Web zum Ausdruck zu bringen. Weiters wird die natürliche Sprache in RDF bzw. WSDL umgewandelt. Um dies zu ermöglichen, wird auf DAML+OIL zurückgegriffen [13].

In Abschnitt 3 wird ein Tool beschrieben, welches die in Abschnitt 1 und 2 definierten konzeptionellen Graphen zueinander in Relation setzt und die semantischen Repräsentationen (RDF bzw. WSDL) miteinander vergleicht. Dadurch wird die Query, die der Benutzer über das User Interface erzeugt und abgesetzt hat, ausgeführt [13].

Der Ansatz von Yao und Etzkorn in [13] beinhaltet weiterhin die Definition einer intelligenten Internetsuchmaschine, die automatisch Software-Komponenten aus dem Internet herunterlädt und dem Analyse-Tool, das in Abschnitt 2 des semantik-basierten Ansatzes in [13] definiert wurde, zur

[3]DAML plus OIL – „DAML-ONT (Darpa Agent Markup Language – Ontology) plus OIL (Ontology Inference Layer)".
[4]OWL – „Web Ontology Language".
[5]RDF – „Resource Description Framework".
[6]WSDL – „Web Service Description Language".
[7]UDDI – „Universal Description, Discovery and Integration".

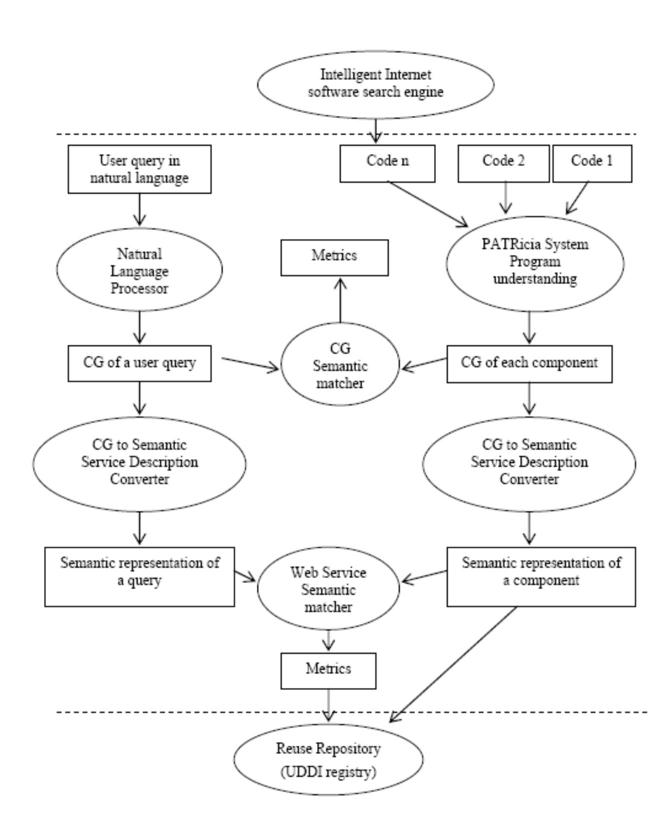


Abbildung 3: Datenflussdiagramm des semantik-basierten Klassifizierungs-Ansatzes (Quelle: [13])

Verfügung stellt, um die Komponenten zu klassifizieren. Wie in Abbildung 3 ferner zu sehen ist, definiert der Ansatz eine weitere Komponente, welche die Suchergebnisse einer Query bei UDDI registriert und somit öffentlich zur Verfügung stellt [13].

Vergleicht man nun diesen Ansatz mit der Arbeit von Prieto-Diaz und Freeman in [11], wird ersichtlich, dass vom facettenorientierten Klassifizierungs-Schema abgewichen wird. Auch der konzeptionelle Graph bekommt bei diesem Ansatz eine andere Bedeutung zugesprochen.

Auch Penix et al. definieren in [9] einen Ansatz, der semantische Eigenschaften verwendet, um Software-Komponenten zu klassifizieren und wiederzufinden.

5.4 Behavior Sampling

Podgurski und Pierce kritisieren in ihren Ansatz in [10], dass bestehende Wiederauffindungsmethoden und Bibliotheken, wie zum Beispiel das Library-System von Prieto-Diaz und Freeman aus [11], Source-Code als puren Text interpretieren und auf dessen Ausführbarkeit, als Unterstützung für Software Reuse, vergessen [10].

Wie allgemein bekannt ist, transformiert Software Input zu Output. Dieser einfachen Eigenschaft macht sich der Ansatz von Podgurski und Pierce in [10] zu nutze: ein Benutzer, in der Literatur als „Searcher" bezeichnet, der auf der Suche nach einer bestimmten Software-Komponente ist, spe-

zifiziert ein Interface für diese Komponente. Dieses Interface muss folgende Eigenschaften definieren [10]:

- Anzahl der Parameter
- Datentypen der Parameter
- Modi der Parameter (Input bzw. Output)

Das Interface, das von einem Benutzer spezifiziert wird, wird von Podgurski und Pierce in [10] als „target-interface specification“ bezeichnet. Anschließend werden für dieses Interface Input-Werte ausgewählt, mit denen die Software-Komponente ausgeführt werden soll. Ebenfalls werden vom Benutzer Output-Werte definiert, die bei Ausführung der Komponente erreicht werden müssen [10].

Der wichtigste Schritt ist die Suche in der Software-Bibliothek: alle durchsuchten Komponenten, die die „target-interface specification“ erfüllen, kommen als mögliche wiederverwendbaren Komponenten in Frage. Dabei müssen jedoch auch die Parametertypen und -modi übereinstimmen [10].

Anschließend werden die ausgewählten Komponenten automatisch mit den Input-Werten, die der Benutzer definiert hat, ausgeführt und die entstandenen Output-Werte mit den Benutzer-Output-Werten verglichen. Jede Komponente, die mit den Benutzerspezifikationen auf diese Weise übereinstimmt, wird dem Benutzer zurückgeliefert [10].

5.5 Kritische Anmerkung

Kritisch anzumerken bzw. zu hinterfragen sei, ob die in der Arbeit diskutierten Ansätze und Technologien [1, 4, 10, 13] das Klassifizierungs-Problem geeignet lösen. Dabei bleiben viele Fragen bezüglich der Implementierung (viele Ansätze definieren ausschließlich die Beschreibung der Problemlösung, jedoch keine konkrete Realisierung) und den konkreten Verwendungen dieser Klassifizierungs-Schemata offen.

Weiters sei kritisch zu bemerken, dass in den diskutieren Lösungen [1, 4, 10, 13] zumeist nur die Klassifizierung von Source-Code, der jedoch nur eines von vielen Artefakten im Software-Entwicklungsprozess darstellt, berücksichtigt wird. In einem modernen (objekt-orientierten) Prozess existiert jedoch eine Vielzahl von anderen Artefakten, man denke zum Beispiel an Dokumente und Ideen, die im Laufe der Requirements- und Design-Phase entstehen.

6. KONKLUSION

Ausgehend von der Arbeit von Prieto-Diaz und Freeman wurde in diesem Artikel die Klassifizierung von Software-Komponenten, die damit verbundenen Probleme und Ansätze zur Lösung dieses komplexen Themas diskutiert.

Prieto-Diaz und Freeman realisierten in [11] das facettenorientierte Klassifizierungs-Schema, welches Grundlage für zahlreiche darauf aufbauende Ansätze und Technologien darstellt und sich heute in diesen wiederspiegelt.

Im Zuge dieser Arbeit wurden einige dieser aufbauenden Ansätze untersucht und zur Arbeit von Prieto-Diaz und Freeman in Relation gesetzt. Dabei wurde speziell auf das Klassifizierungs- und Wiederauffindungs-Problem eingegangen.

Abschließend bleibt zu sagen, dass die untersuchten Probleme des Software Reuse auch in Zukunft diskutiert werden sollten, um diese so gut als möglich zu lösen und die Wiederverwendung bestehender Komponenten in eine noch zentralere Position der Softwareentwicklung zu rücken.

7. LITERATUR

[1] J. Boerstler. Feature oriented classification for software reuse. In *7th International Conference on Software Engineering and Knowledge Engineering*, pages 204–211, 1995.

[2] B. Coulange. *Software Reuse*. Springer, 1998.

[3] I. Herman. Semantic web, http://www.w3.org/2001/sw/, letzter Zugriff: 17.05.2007.

[4] E.-A. Karlsson, S. Sorumgard, and E. Tryggeseth. Classification of object-oriented components for reuse. In *Proceedings of the Conference on Technology of Object-Oriented Languages and Systems (TOOLS 7)*, pages 21–31, 1992.

[5] Ch. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[6] H.-C. Liao and F.-J. Wang. Software reuse based on a large object-oriented library. *ACM Sigsoft, Software Engineering Notes*, 18(1):74–80, 1993.

[7] Y. Matsumoto. A software factory: An overall approach to software production. In *Freeman P.: Tutorial Software Resuability; IEEE.CS POress*, pages 155–178, 1987.

[8] A. Mili, R. Mili, and R. T. Mittermeir. A survey of software reuse libraries. *Annals of Software Engineering*, 5(0):349–414, 1998.

[9] J. Penix, P. Baraona, and P. Alexander. Classification and retrieval of reusable components using semantic features. In *The 10th Knowledge-Based Software Engineering Conference*, pages 131–138, 1995.

[10] A. Podgurski and L. Pierce. Retrieving reusable software by sampling behavior. *ACM Transactions on Software Engineering and Methodology*, 2(3):286–303, 1993.

[11] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, 1987.

[12] R. Rada. *Software Reuse: Principles, Methodologies and Practices*. Intellect, 1995.

[13] H. Yao and L. Etzkorn. Towards a semantic-based approach for software reusable component classification and retrieval. In *Proceedings of the 42nd annual Southeast regional conference, SESSION: Software engineering Nr. 1*, pages 110–115, 2004.

Software-Entwicklungsprozesse

Markus Schneider
0460242
m2schnei@edu.uni-klu.ac.at

Abstract

Kostenminimierung, Qualitätssoftware und maximaler Nutzen sind die Hauptziele der Softwareentwicklung. Um diese Ziele zu erreichen bedarf es methodischer Vorgehensweisen im Entwicklungsprozess. Die Schlüsselkomponente dafür sind Prozessmodelle. Mit dem Wasserfallmodell begann die Ära strukturierter Entwicklungsprozesse. Bis heute haben sich zahlreiche Vorgehensmodelle entwickelt und etabliert. Der auf dem Wasserfallmodell basierende Ansatz findet sich in den meisten Prozessmodellen wieder.

Ausgehend von Boehms Artikel „Software Engineering", beschreibt das vorliegende Paper die bekanntesten Entwicklungsansätze und gibt einen Einblick in klassische Prozessmodelle. Näher wird im Artikel auf den schwergewichtigen Rational Unified Process und die agile Methodik eXtreme Programming eingegangen.

1. Einleitung

Mitte der sechziger Jahre steigerte sich die Leistung von Hardware enorm. Software die im Stande war die Ressourcen auszunutzen wurde benötigt. Fehlendes Wissen über Methoden, mangelndes Fachpersonal und fehlerhafter Code führten zu kostenineffizienten Produktionen von Software. Um deren Entwicklung zu verbessern, sponserte das NATO SCIENCE COMITTE 1968 in Garmisch eine Konferenz. Sie war die Geburtsstunde des Terminus „Software-Engineering" und behandelte Standardisierungen für Design und Produktion [3].

Bis heute besteht die Schwierigkeit zuverlässige Softwaresysteme termingerecht und kosteneffizient zu erzeugen. Laut CHAOS Report (1994, 2002, 2004) der Standish Group werden im Mittel nach wie vor 74 % der Projekte nicht erfolgreich abgeschlossen. Davon überschreiten 52 % den Zeit- und Budgetrahmen, 22 % werden frühzeitig abgebrochen und lediglich 26 % der Projekte schließen zufrieden stellend ab [7, 8]. Boehm bekräftigt aber, dass der Grund für einen Projektabbruch nicht ausschließlich am schlechten Projektmanagement liegt [21]. Vielmehr führen Gründe wie schlechte oder unvollständige Anforderungen, mangelnde Planung oder Wirtschaftlichkeitsfaktoren zu einem Projektabbruch. Die nüchterne Statistik verdeutlicht wie notwendig es ist, Softwareentwicklung diszipliniert und mit klaren Vorgehensweisen zu betreiben.

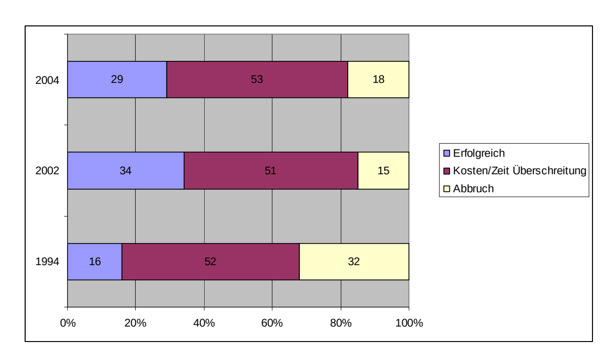


Abb. 1: CHAOS Report (1994, 2002, 2004) (Datenquelle: [7, 8])

Software-Engineering kann als geschichtete Technologie aufgefasst werden, und gliedert sich in die vier Schichten: Werkzeuge, Methoden, Prozesse und eine Schicht für den Qualitätsfokus [14]. Das Fundament für Software-Engineering bildet die Prozessebene. Die impliziten Softwareprozesse sind nach Pressman der Leim, der die Technologieebenen zusammenhält [14]. Prozessmodelle[1] ermöglichen eine vernünftige und zeitgerechte Entwicklung von Softwareprodukten (vgl. [14] pp. 22). Sie sind Schlüsselkomponenten für die methodische Entwicklung von Softwareprodukten und leisten einen enormen Beitrag zur Verbesserung des Entwicklungsprozesses sowie der Produktqualität.

[1] Die Begriffe Prozessmodell und Vorgehensmodell werden im Paper synonym verwendet

Winston Royce veröffentlichte 1970 das Wasserfallmodell. Dieses gilt als erstes konkretes Vorgehensmodell und erlangte durch Barry Boehms Artikel „Software-Engineering“ (1976) [1] breite Bekanntheit. Boehms Paper [1] ist der Leitartikel für die vorliegende Arbeit, da das Wasserfallmodell die Basis für die meisten Prozessmodelle bildet [10]. Im weiteren Verlauf, zeigt dieses Paper einen Einblick in die Entwicklung von Prozessmodellen. Es werden Entwicklungsansätze beschrieben und Vorgehensmodelle diskutiert, die seit der Etablierung des Wasserfallmodells entstanden sind. Bei den bekanntesten Prozessmodellen findet sich eine Unterscheidung in deren Ansätzen. Dies sind unter anderem der evolutionäre und inkrementelle Ansatz, die iterative und phasenorientierte Entwicklung, sowie die moderne Entwicklung mit Wiederverwendung. Kapitel 2 beschreibt diese Ansätze und gibt einen Auszug klassischer Prozessmodelle. Kapitel 3 und 4 erläutern den schwergewichtigen Rational Unified Process und die leichtgewichtige Methodik eXtreme Programming als Vertreter moderner Prozessmodelle.

2 Allgemeine Prozessmodelle

Bei einem Prozess handelt es sich um einen Rahmen, der Aktivitäten, Methoden und Verfahren für die Herstellung und Überprüfung von Softwareprodukten beschreibt [2]. Prozessaktivitäten können sich laut IEEE Standard 610.12 (1990) überlappen und sie können mehrmals wiederholt werden. Angelehnt an die Beschreibung des Wasserfallmodels gibt es vier grundlegende Prozessaktivitäten, die sich in den gängigen Prozessmodellen wieder finden: Dies sind Spezifikation, Design, Implementierung und Test.

Prozessmodelle sorgen für Transparenz und Steuerbarkeit im Entwicklungsprozess. Sie legen Standards und Richtlinien fest, definieren Verantwortlichkeiten und Kompetenzen, legen die Reihenfolge von Arbeitsschritten fest und koordinieren Prozessaktivitäten und Ergebnisse. Pankaj Jalote definiert ein Prozessmodell wie folgt:

„A development process model specifies some activities that, according to the model, should be performed, and the order in which they should be performed“ [4].

2.1 Entwicklungsansätze

Phasenorientierte Entwicklung (Lineare Prozesse). Die phasenorientierte Herangehensweise teilt die Projektaktivitäten in gleichnamige Phasen ein. In jeder Phase werden Ergebnisse in Form von Dokumenten produziert, die als Input in der Folgephase benötigt werden. Am Ende jeder Phase werden Teilziele, die so genannten Meilensteine, erreicht, an welchen der Projektverlauf messbar ist. Eine Phase kann somit als Zeitspanne zwischen zwei Meilensteinen aufgefasst werden. Grundsätzlich gilt, dass sich beim phasenorientierten Entwicklungsansatz die einzelnen Teilphasen nicht überlappen, und jede Phase abgeschlossen sein muss, bevor die nächste startet. Bekannte Vorgehensmodelle mit phasenorientiertem Ansatz sind das klassische Wasserfallmodell, das Phasenmodell und das V-Modell.

Iterative Entwicklung. Beim iterativen Ansatz werden die Prozessaktivitäten in Iterationsrunden mehrmals ausgeführt. Einsatz findet diese Herangehensweise häufig bei Systemen, in welchen die Anforderungen von Beginn an nicht vollkommen klar sind, oder aber bei großen Systemen, die nicht in einem Zug entwickelt werden können. Jede Iteration kann als „Teilprojekt“ aufgefasst werden, wobei nach Iterationen ein Gesamtsystem entsteht. Mit dem Spiralmodell als bekannten Vertreter dieser Gruppierung, setzt der iterative Entwicklungsansatz auf die frühestmögliche Reduzierung der Projektrisiken. Iterative Softwareentwicklung bietet einen Lösungsansatz um adäquat auf Änderungen in den Kundenanforderungen reagieren zu können. Sie findet häufig bei modernen Prozessmodellen Einsatz.

Inkrementelle Entwicklung. Im Gegensatz zum iterativen Ansatz, der in jedem Iterationsschritt das Produkt verfeinert bis ein Gesamtsystem entsteht, werden bei der inkrementellen Entwicklung in Zyklen neue Systemteile (Inkremente) erstellt und in einem Kernsystem integriert. Voraussetzung ist, dass es sich beim Gesamtsystem um ein offenes System handelt, welches in Ausbaustufen realisiert wird (vgl. [6] pp. 164). Die inkrementelle Entwicklung hat den enormen Vorteil, dass bereits fertige Systemteile an den Kunden geliefert werden können. Der Kunde ist somit frühzeitig in der Lage, Tests durchzuführen und ein Feedback zu erstellen. Aufgrund der Tatsache, dass Systemteile mit hoher Priorität als erstes entwickelt und ausgeliefert werden, reduzieren sich mit diesem Ansatz auch die Projektrisiken (vgl. [5] pp. 102).

Evolutionäre Entwicklung. Diese Vorgehensweise erweist sich als hilfreich, wenn die Spezifikation nicht ausreichend genug ausgearbeitet werden konnte oder der Auftraggeber keine klare Vorstellung vom Endprodukt hat. Die Prozessaktivitäten finden in diesem Ansatz parallel statt und tauschen gegenseitig Informationen aus. Grundsätzlich werden zwei Richtungen in der evolutionären Entwicklung

unterschieden. Erstens die explorative Entwicklung mit dem Ziel, in Kooperation mit dem Kunden die exakten Systemanforderungen herauszufinden. Zweitens anhand eines Wegwerf-Prototypen die Kundenbedürfnisse zu ermitteln. Der Vorteil dieser Methode liegt in der schrittweisen Erstellung der Spezifikation. Der Ansatz hat aber auch zwei gravierende Nachteile. Zum einen ist der Prozess nicht sichtbar, wodurch die Kontrollfähigkeit für die Geschäftsführung nicht gegeben ist. Zum anderen führt die Methode zu unstrukturierten Systemen. (vgl. [5] pp. 98 – 99)

Entwicklung mit REUSE. Heutige Systeme werden verstärkt durch wieder verwendete Komponenten erzeugt. Nach Boehm steigt die Anzahl an LOCS[2] in den Systemen, gleichzeitig sinken aber die Kosten für eine LOC [9]. Es zeigt sich der Trend Software mit bestehenden oder angekauften COTS[3] Komponenten zu erzeugen. In den letzten Jahren hat sich für diesen Prozess das CBSE[4] entwickelt. CBSE ist ein neuer Entwicklungsansatz, der die Einbindung von Softwarekomponenten berücksichtigt. Das CBSE – Prozessmodell unterscheidet sich von herkömmlichen Prozessmodellen, wie beispielsweise dem Wasserfallmodell, in den Stufen zwischen Anforderungsspezifikation und Systemvalidierung [5]. Diese Zwischenstufen sind nach [5]: Analyse der Komponenten, Anpassung der Anforderungen, Systementwurf mit Wiederverwendung, Entwicklung und Integration von COTS- und eigenen Komponenten.

2.2 Auszug klassischer Prozessmodelle

Wasserfallmodell. In der klassischen Ausführung ist das Wasserfallmodell ein rein phasenorientiertes Vorgehensmodell. Durch die Rückwärtsverkettung der einzelnen Phasen bekommt es aber einen iterativen Charakter. Seine größten Probleme liegen in der strikten Abgrenzung der einzelnen Phasen. In nahezu allen Softwareprojekten kommt es zu unvorhersehbaren Änderungen, wodurch es zur Notwendigkeit kommt, frühere Phasen abermals zu durchlaufen. Das Modell geht jedoch von vollständigen Anforderungen aus und lässt lediglich kleine Iterationen zwischen benachbarten Phasen zu, wodurch es beispielsweise keinen Weg vom Betrieb zurück in die Entwicklungsphase gibt [4, 6]. Weiters fordert die dokumentengetriebene Vorgehensweise formale Dokumente am Ende jeder Phase. Diese werden aber nicht bei allen Projekten benötigt wodurch nur unnötiger Overhead entsteht [4].

Spiralmodell. Beim Spiralmodell handelt es sich um ein generisches Modell. Es wurde 1988 von Barry Boehm beschrieben und entstand laut ihm über Jahre hinweg aufgrund diverser Verfeinerungen am Wasserfallmodell [10]. Spiralförmig durchlaufen die Prozessaktivitäten in mehreren Zyklen vier konkreter Quadranten. Jeder Zyklus beginnt im ersten Quadranten wo Ziele, Alternativen und Randbedingungen ermittelt werden. Das Spiralmodell ist ein risikogetriebenes Vorgehensmodell um Projektrisiken frühzeitig erkennen zu können und dem größten Risiko aus dem Weg zu gehen. Risikoanalyse und Evaluierung von Alternativen finden im zweiten Quadranten statt. Danach wird entschieden, ob mit einem Prototypingansatz, einem simulationsorientieren Vorgehen oder anderen Arten von Vorgehensweisen fortgefahren wird. Der dritte Quadrant ist für die Entwicklung und Tests der Teilprodukte verantwortlich. Im vierten und letzten Quadranten wird jeweils die nächste Phase geplant.

Initial startet der Prozess mit einer Machbarkeitsstudie in Runde 0. Hier wird entschieden, ob das Projekt in Angriff genommen wird oder nicht. In Runde 1 werden ein Projektplan und ein Konzept für den Projektablauf erstellt. Weiters werden eine Kostenanalyse erzeugt und die Projektziele detaillierter spezifiziert. Runde 2 ist für die Erfassung der Anforderungen auf höchster Ebene. In den nachfolgenden Runden wird das Produkt gefertigt, wobei die Aktivitäten Design, Implementierung und verschiedene Arten von Tests durchgeführt werden. (vgl. [10])

3 Der Rational Unified Process (RUP)

In den 80er Jahren erfuhren objektorientierte Sprachen breite Popularität. Für die Entwicklung von OO-Softwaresystemen schmiedeten Grady Booch, James Rumbaugh und Ivar Jacobson in den 90er Jahren den Unified Process (1999) [12]. Booch, Rumbaugh und Jacobsen alias die „Tres Amigos“ entwickelten zuvor die OO[5] Techniken OOA[6]/OOD[7], OMT[8] und OOSE[9], welche zusammen mit dem Anwendungsfallmodell die Basis für den Unified Process darstellen. Der Prozess an sich ist keine konkrete Ausprägung sondern stellt ein Metamodell für Software Prozesse dar. Durch die Fusion der Rational Software Corporation und Jacobsons Objectory AB (1995),

[2] LOCS – Lines of Code
[3] COTS – Commercial off the Shelf
[4] CBSE – Component Based Software Engineering

[5] OO – Objekt-orientiert
[6] OOA – Objekt-orientierte Analyse
[7] OOD – Objekt-orientiertes Design
[8] OMT – Object-modeling Technique
[9] OOSE – Objekt-orientiertes Software Engineering

entstand 1998 der Rational Unified Process. Der RUP ist im Grunde genommen ein Produkt vom Hause Rational das ständig weiterentwickelt wird. Er ist ein werkzeugzentrierter Prozess und eng mit der UML[10] verbunden. Der RUP ist eine konkrete Implementierung des abstrakten Unified Process und ein Paradebeispiel eines modernen Vorgehensmodells [6].

3.1 Strukturen

Das Wesen des RUP liegt in den fünf Prozessbausteinen: Tätigkeiten, Workflows, Iterationen, Phasen und Zyklen. Der RUP besteht aus einer dynamischen und statischen Perspektive. Die dynamische Perspektive schließt den zeitlichen Verlauf anhand von Zyklen, Phasen, Iterationen und Meilensteinen ein. Aktivitäten, Artefakte (Ergebnisse), Workflows und Mitarbeiter bilden den statischen Teil des Models. Die statische Struktur legt fest, wer, was, wie und wann zu tun hat.

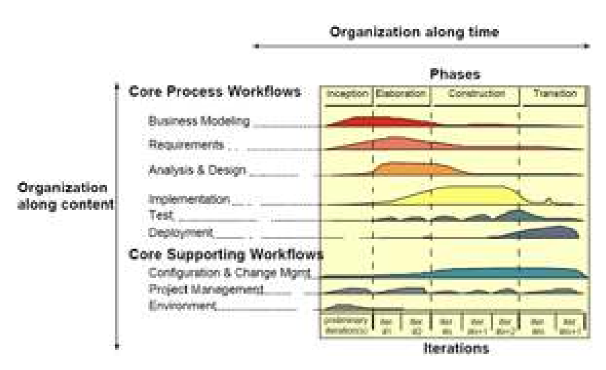


Abb. 2: Struktur des RUP (Quelle: [13])

Tätigkeiten. Jeder Projektmitarbeiter schlüpft in eine oder mehrere Rollen, wodurch sich seine Verantwortlichkeiten im Projekt konzentrieren. Die verschiedenen Tätigkeiten („Activities“) stellen wohldefinierte Arbeitseinheiten dar und werden von Mitarbeitern in deren Rollen durchgeführt. Tätigkeiten haben das Ziel Artefakte zu erstellen. Als Artefakt versteht man ein konkretes Ergebnis, dass von einem oder mehreren Mitarbeitern produziert wird. So ein Ergebnis ist beispielsweise die Realisierung eines Anwendungsfalls.

Workflows. Workflows definieren die Abfolge von verschiedenen Tätigkeiten. Ein Workflow kann somit als Partition von Rollen und Tätigkeiten aufgefasst werden. Im RUP existieren sechs Kernworkflows und drei unterstützende Workflows. Die Kernworkflows sind Business Modelling, Requirements, Analysis & Design, Implementation, Test und Deployment. Bis auf den Deployment Workflow erstrecken sich alle Kernworkflows über die gesamten vier Projektphasen. Als unterstützende Workflows nennt Kruchten Configuration and Change Management, Project Management und Environment [11].

[10] UML – Unified Modeling Language

Iterationen. Eine Iteration fasst verschiedene Workflows zusammen und durchläuft die Prozessaktivitäten von der Anforderungsspezifikation bis zur Evaluierung. Jede Iteration resultiert in einem internen Release der Software und treibt die Entwicklung schrittweise voran.

Phasen. Das Modell gliedert sich in die vier Phasen:

1. Beginn (engl. Inception)
2. Ausarbeitung (engl. Elaboration)
3. Konstruktion (engl. Construction)
4. Auslieferung (engl. Transition).

Jede Phase kann wiederum aus mehreren Iterationen bestehen und liefert verschiedene Ergebnisartefakte. Am Ende jeder Phase wird die nächste Phase geplant und ein bestimmter Meilenstein muss erreicht sein. Nach Kruchten [11] haben die vier Phasen unterschiedliche Kapazitäten in Bezug auf Aufwands- und Zeitanteil.

Phase	Zeit	Aufwand
Beginn	10%	5%
Ausarbeitung	30%	20%
Konstruktion	50%	65%
Auslieferung	10%	10%

Tab. 1: Relatives Gewicht der Phasen

Ziel der Beginnphase ist es eine Machbarkeitsstudie zu entwickeln und sich über Projektumfang und Randbedingungen Gedanken zu machen. Es werden die kritischen Anwendungsfälle erfasst und Projektrisiken frühzeitig identifiziert. Für die wirtschaftliche Betrachtung werden Kosten- und Zeitabschätzungen angestellt. Als Ergebnisartefakte liefert die Beginnphase eine Produktvision, ein Glossar, eine Risikenliste, ein Überblicksmodell der Anwendungsfälle, einen groben Projektplan und einen Business Case. Nach Abschluss dieser Phase ist der Lifecycle Objective Meilenstein erreicht, welcher auf spezielle Kriterien hin evaluiert wird. (vgl. [13])

In der Ausarbeitungsphase gilt es, die Anwendungsfälle nahezu vollständig zu erfassen, eine fundierte Systemarchtiketur zu entwerfen sowie ein Verständnis für den Problembereich zu entwickeln. Die Ergebnisar-

tefakte in dieser Phase sind ein ausführbarer Architekturprototyp und eine Architekturbeschreibung, ein zu ca 80% fertiges Anwendungsfallmodell, die Ausformulierung der nichtfunktionalen Anforderungen, sowie eine aktualisierte Version des Projektplans und der Risikenliste. Am Ende dieser Phase ist der Lifecycle Architecture Meilenstein erreicht und bereit, evaluiert zu werden. (vgl. [13])

Entwurf und Implementierung sind der Fokus der Konstruktionsphase. Am Ende dieser Phase sollte ein fertiges Beta-Release erstellt sein, welches fähig für die Weitergabe an den Kunden ist. Weitere Artefakte sind ein Benutzerhandbuch, die Systemdokumentation und eventuelle Schulungsunterlagen. Mit Beendigung der Konstruktionsphase ist der Initial Operational Capability Meilenstein erreicht. (vgl. [13])

Die letzte Phase des Entwicklungszyklus ist die Auslieferungsphase. In ihr wird eine Installationsroutine erstellt und das fertige System an den Kunden übergeben. Dort wird es in seiner Zielumgebung installiert und vom Kunden auf Akzeptanz getestet. Sollten im Akzeptanztest kleine Mängel festgestellt werden, wird das Produkt nachgebessert. Falls es gefordert wird, werden die Benutzer beim Kunden auf das System eingeschult und die vollständigen Dokumente übergeben. Mit dem Erreichen des Product Release Meilenstein wird das Projekt abgeschlossen. (vgl. [13])

Zyklen. Unter einem Zyklus versteht man den gesamten Durchlauf der vier Phasen. Wird eine neue Version der Software gefordert, so entsteht diese in einem neuerlichen Zyklus.

3.2 Charakteristiken

Der RUP wird von Kruchten, dem damaligen Leiter für die Entwicklung des Prozessmodells, als iterativ, anwendungsfallgesteuert, architekturzentriert und risikoorientiert charakterisiert [11]. Im RUP finden sich zahlreiche Entwicklungsansätze aus Kapitel 3 wieder. Der Wasserfallansatz, Phasenmodell, iterative, inkrementelle und evolutionäre Entwicklung sind Teil des RUP.

Anwendungsfallgesteuert. Die Grundlage für viele Tätigkeiten bilden die Anwendungsfälle. Anhand dieser werden die funktionalen Anforderungen erhoben. Sie beschreiben die Funktionalität des Systems aus Benutzersicht, werden für den Entwurf von Testfällen herangezogen und beinhalten die Basisinformationen für den Architekturentwurf.

Architekturzentriert. Im Prozess wird Architektur frühzeitig entwickelt und laufend verbessert. Die Architektur kann als „Skelett" des Systems aufgefasst werden und bildet das Fundament des Produkts. Für die Betrachtung der Systemarchitektur aus unterschiedlichen Blickwinkeln, stellt der RUP fünf Sichten zur Verfügung. Die logische Sicht abstrahiert das Entwurfsmodell. Sie identifiziert Pakete, Subsysteme und Klassen, wobei keine technischen Details mit einfließen. Diese Sicht beleuchtet ausschließlich fachliche Bestandteile und ist für den Endanwender und das Management geeignet. Implementierungs-, Prozess- und Verteilungssicht decken das technische Architekturmodell ab. Die Implementierungssicht ist für den Programmierer. In ihr werden die Softwaremodule organisiert. Der Verantwortungsbereich für nebenläufige und parallele Aspekte des Systems liegt bei der Prozesssicht. Sie ist für Systemintegratoren geeignet und deckt weiters noch Performanz und Skalierbarkeit ab. Systemingenieure befassen sich mit der Verteilungssicht. In ihr betrachtet man die physische Verteilung der Systemkomponenten auf verschiedene Rechnerknoten und befasst sich mit Installation und Auslieferung. Die Anwendugsfallsicht umfasst die Use Cases. Diese Sicht ist die zentralste aller Sichten, da die Use Cases das Fundament für sämtliche Aktivitäten sind. Die Anwendungsfallsicht überlappt sich mit allen anderen vier Sichten, weil diese die Informationen aus der Anwendungsfallsicht benötigen.

Iterativ / Inkrementell. Der iterative und inkrementelle Aspekt vom RUP geht bereits aus Punkt 3.1 hervor. Aufbauend auf dem Spiralmodell [10] von Boehm läuft die technische Umsetzung in Iterationen ab, wodurch Arbeitsergebnisse schrittweise weiterentwickelt werden. Wasserfallartig werden die Prozessaktivitäten pro Iteration abgearbeitet. Am Ende jeder Iteration wird die nächste Iterationsrunde vorausgeplant. Die Anwendung mehrerer Zyklen erlaubt die inkrementelle Produktentwicklung, wodurch mehrere Versionen der Software produziert werden können [6].

Risikoorientiert. Durch die frühzeitige Adressierung der Risiken versucht man die kritischen Aspekte des Projekts zu einem Zeitpunkt zu erfassen, an dem noch genügend Zeit für den restlichen Projektverlauf übrig ist. Ziel ist es, problematische Systemteile schnell zu realisieren, um die Risiken zu minimieren.

Vor- und Nachteile. Aufgrund seines bürokratischen Charakters ist der RUP als ein schwergewichtiger Entwicklungsprozess bekannt. Schwergewichtig in dem Sinn, dass der Prozess eine strikte Einhaltung der Phasen und Kernworkflows fordert, wodurch ein hoher

Grad an Verwaltungs- und Dokumentationsaufwand entsteht. Genau genommen stellt der RUP aber ein umfassendes Framework für einen Entwicklungsprozess dar, der an die jeweiligen Bedürfnisse angepasst werden muss. Laut Carsten Dogs und Timo Klimmer wird dem Prozessmodell dadurch eine gehörige Portion des Schwergewichtes genommen [18]. Ein enormer Vorteil dieses Modells liegt im Überblick und in der Koordinationsmöglichkeit für die Projektleitung. Bei kleinen Entwicklerteams ist der RUP eher nicht zu empfehlen, weil die geforderte Rollenverteilung nicht gestaltet werden kann. An die Entwicklungs-organisation bestehen für den Einsatz vom RUP mehrere Vorraussetzungen. Dies sind nach Ludewig und Lichter in [6]: Ausgezeichnete Konfigurations- und Änderungsmöglichkeiten, Gute Projektmanage-ment-Fertigkeiten und Kenntnisse objektorientierter Konzepte. Die Stärken des Prozessmodells liegen in der guten Darstellung und dem hohen Detaillierungs-grad des Prozesses. Es eignet sich aber eher für große Softwareprojekte. Folgende Problemebereiche ergeben sich mit dem RUP: Schwierige Anpassung in einer Organisation, instabile Prozessdefinition aufgrund ständiger Weiterentwicklung der Prozesskonzepte und die Vortäuschung, dass die Sofwareentwicklung algorithmisch durchgeführt werden kann (vgl [6] pp 198 – 199).

4 Agile Softwareentwicklung

Agile Methodiken sind als leichtgewichtige Entwicklungsprozesse bekannt, da hier auf alles verzichtet wird, was keine Relevanz für das Projekt hat. Die agile Bewegung entwickelte sich in den späten 90er Jahren aufgrund der Tatsache, dass schwergewichtige Prozesse einen hohen büro-kratischen Aufwand nach sich zogen, wenn sich Änderungen ergaben [9]. Bei der agilen Entwicklung steht kein strikter Prozess im Mittelpunkt, sondern der Mitarbeiter. Im Manifest der Agile Alliance werden vier Werte genannt, welche bei einer agilen Vorgehensweise ständig beachtet werden müssen. Dies sind:

- Menschen und Zusammenarbeit vor Prozessen und Werkzeugen
- Funktionierende Software vor umfangreicher Dokumentation
- Zusammenarbeit mit dem Kunden vor vertraglicher Verhandlung
- Reaktion auf Veränderung vor Einhaltung eines Plans

Aus dem ersten Wert geht hervor, dass der Programmierer im Zentrum der Entwicklung steht, da nach Ansicht der Agile Alliance das Entwicklerteam der Erfolgsfaktor für gelungene Projekte ist. Wichtig ist aber die Kommunikation zwischen den Entwicklern untereinander. Es hat keinen Sinn, zehn sehr gute Programmierer im Team zu haben, die nicht teamfähig sind. Umfangreiche Dokumentation erfordert einen hohen Zeitaufwand und es ist schwierig, diese während der Entwicklungsphase aktuell zu halten. Der zweite Wert drückt aus, dass keine Dokumente erzeugt werden sollen, für die es keinen unmittelbaren Nutzen gibt. Dem dritten Wert kann entnommen werden, dass es besser ist, in ständiger Kooperation mit dem Kunden die Anforderungen des Systems auszuarbeiten, als anfänglich eine Spezifikation zu schreiben und diese als endgültige Basis und Vertrag für die Erstellung des Systems heranzuziehen. Solch ein Ansatz führt meistens zu Produkten niedriger Qualität. Viele Softwareprojekte scheitern aufgrund von Änderungen in den Systemanforderungen. Um diesem Problem zu entgehen gibt es den vierten Wert. Dieser beschreibt die Flexibilität agiler Vorgehensweisen. (vgl. [16])

4.1 eXtreme Programming (XP)

eXtreme Programming ist der wohl bekannteste und am meisten eingesetzte Vertreter agiler Entwicklungsmethoden. Dogs und Klimmer gliedern die agilen Methodiken in die vier Kategorien: prozessorientierte, mitarbeiterzentrierte, werkzeug-zentrierte und unvollständige Methodiken [18]. eXtreme Programming wird den prozessorientierten Methodiken zugeordnet, da es sich stark auf einen iterativen Prozess konzentriert und somit in diesem Paper einen passenden Vertreter agiler Methodiken darstellt. Im Gegensatz zu herkömmlichen iterativen Prozessen wie beispielsweise dem Spiralmodell, erlaubt XP extrem kurze Iterationsschritte. Pro Iterationsrunde werden Anforderungen ermittelt, der Entwurf wird laufend verbessert, Systemteile werden implementiert und getestet. Bei XP werden jedoch die prozessspezifischen Techniken in jeder Aktivität angewandt. Beispielsweise werden die Anforderungen im so genannten Planungsspiel ermittelt. Nach dem Motto der agilen Werte stellt XP die Entwickler und den Kunden in den Mittelpunkt. Im Zentrum von XP wird das Programmieren als Schlüsseltätigkeit dargestellt, wodurch die Methodik eine Reihe konkreter Praktiken für die Implementierung bietet. Kent Beck definiert eXtreme Programming als eine leichte, effiziente, risikoarme, flexible, kalkulierbare, exakte und vergnügliche Art und Weise der Softwareentwicklung [17]. In Zusammenarbeit mit Ward Cunningham und Ron Jeffries entwickelte Kent Beck 1996 die neue Vorgehensweise XP. Auslöser für die Entwicklung war ein Projekt beim Chrysler Konzern, welches am Rande

des Abbruchs stand. Das Wort „eXtreme“ in XP bedeutet, dass alle Prozessaktivitäten in extremer Weise stattfinden. Wenn beispielsweise Testen gut ist, wird in XP ständig getestet. Den Grundpfeiler bei XP bilden vier Werte:

- **Kommunikation –** Forderung intensiver Kommunikation im Team.
- **Einfachheit –** Sämtliche Systemteile müssen so einfach wie möglich entworfen werden.
- **Feedback –** Unterstützt durch die XP Praktiken sollen Mitarbeiter rasches Feedback für ihre erzielten Ergebnisse erhalten.
- **Mut –** Die Entwickler müssen den Mut aufbringen, neue Alternativen heranzuziehen. Sie müssen in der Lage sein, bestehenden Code wegzuwerfen und Teile neu zu entwickeln.

Abgeleitet aus den vier Werten nennt Kent Beck die fünf Grundprinzipien „Unmittelbares Feedback“, „Einfachheit anstreben“, „Inkrementelle Veränderungen“, „Veränderungen wollen“ und „Qualitätsarbeit“ [17]. Neben diesen fünf existieren noch zehn weitere Sekundärprinzipien, die ständig Beachtung finden müssen, wenn XP angewandt wird. Für die praktische Umsetzung eins Projekts gibt es ursprünglich zwölf Techniken. In aufzählender Reihenfolge sind dies: das Plannungsspiel, kurze Releasezyklen, Metapher, einfaches Design, Testen, Refactoring, Programmieren in Paaren, gemeinsame Verantwortlichkeit, fortlaufende Integration, 40-Stunden-Woche, Kunde vor Ort und Programmierstandards. Damit der Rahmen des Papers nicht gesprengt wird, verweise ich für Details zu den XP Praktiken auf Kent Beck [17]. Lediglich die Techniken Plannungsspiel, Pair Programming und Testen möchte ich kurz diskutieren.

Plannungsspiel. In XP werden die Anforderungen mit Story Cards ermittelt. Diese werden vom Kunden verfasst und priorisiert. Sie beinhalten Szenarien für die Systemfunktionalität. Daraufhin gestalten Kunde und Entwicklerteam einen Releaseplan. Die Ziele für jedes Release werden mit dem Kunden (bzw. Kundenvertreter) vor Ort diskutiert. Dadurch wird versucht, das Risiko einer Fehlentwicklung zu kompensieren. [20]

Pair Programming. Beim Pair Programming implementieren zwei Entwickler eine Problemstellung, wobei einer der beiden den Code schreibt und der andere parallel den Code überprüft und einen Schritt voraus denkt. Die Paare mischen sich in zyklischer Reihenfolge durch und arbeiten immer an unterschiedlichen Komponenten. Dadurch wird verhindert, dass Projektwissen durch den Ausfall eines Mitarbeiters verloren geht [20]. Oft wird die Wirtschaftlichkeit von Pair Programming kritisiert. Nach einer Studie von Cockburn und Williams, verschlingt diese Praktik lediglich 15% mehr Kosten [19]. Als Ausgleich dafür fördert es die Qualität von Design, Code und Entwicklerfähigkeiten und reduziert Fehler und Risiken. Laut Cockburn und Williams wird die Fehlerrate um 15% gesenkt [19]. Nach Boehm kostet es um ein Zehnfaches weniger, einen Fehler in der Implementierungsphase auszubessern, als wenn er erst im Betrieb entdeckt wird [1].

Testen. eXtreme Programming adaptiert den TDD[11] - Ansatz, wobei zuerst die Unit-Tests geschrieben werden, bevor das eigentliche Modul entwickelt wird. Die TDD Technik hat den Vorteil, dass sich der Entwickler im Vorhinein Gedanken über die Funktionalität des Moduls macht und durch den Unit-Test sofortiges Feedback bekommt. Dadurch werden Fehler frühzeitig entdeckt und eliminiert. Laut Boehm ist dies essentiell, um Kosten zu sparen. Beim Wasserfallmodell finden die Tests erst nach der Implementierung statt [1]. Dadurch werden 40 – 50 % der Entwicklungskosten für die Nachbearbeitung des Codes geschluckt [1]. Werkzeuge, wie beispielsweise JUnit, unterstützen die Testaktivitäten. Neben den Unit-Tests finden in XP noch die Akzeptanztests statt, bei welchen das Endprodukt vom Kunden auf Funktionalität geprüft wird.

Rollen in XP. Im Prozessmodell wird eine Reihe von Rollen definiert, um die Verantwortlichkeit der Mitarbeiter im Projekt zu konzentrieren. Damit wird verhindert, dass Entscheidungen von nicht qualifizierten Personen getroffen werden. Wie bereits einleitend hervorgeht tragen der Programmierer und der Kunde die wichtigsten Rollen in einem XP – Projekt. Neben diesen beiden gibt es noch den XP-Trainer, den Projektleiter, den Technologieberater, den Tester und den Tracker. Für Details zu den einzelnen Personenrollen sei auf [17, 18] verwiesen.

Einschränkungen von XP. eXtreme Programming kann nicht bei jedem Projekt eingesetzt werden. Es setzt voraus, dass es sich beim Entwicklerteam um äußerst erfahrene Personen handelt und das Team nicht mehr als 15 Leute umfasst. Dadurch, dass Kommunikation einen entscheidenden Wert in der Prozessphilosophie darstellt, dürfen die Entwickler nicht örtlich getrennt arbeiten. XP muss in die Unternehmenssituation passen, damit den Mitarbeitern der nötige Freiraum zur Verfügung steht. Weiters ist XP nicht für

[11] TDD – Test Driven Development

Projekte zu empfehlen, bei welchen detaillierte Dokumentation oder ein festgelegter Systementwurf gefordert ist.

5. Zusammenfassung

Die Anwendung von Prozessmodellen in der IT-Branche ist nicht mehr wegzudenken. Prozessmodelle sind der Wegweiser zum erfolgreichen Projektabschluss. Schwierig gestaltet sich allerdings die Einbindung eines Prozessmodells in ein Unternehmen. Es muss die jeweilige Leistung des Vorgehensmodell betrachtet und abgewogen und für das Projekt und die Unternehmenssituation angepasst werden. Die zwei Prozessmodelle RUP und XP verfolgen das gleiche Ziel: die Erstellung objektorientierter Software. RUP und XP setzen auf einem iterativen Entwicklungsansatz auf. In beiden Modellen findet sich der Wasserfallansatz wieder. Der RUP als schwergewichtiger Prozess gleicht die Komplexität der Softwareentwicklung durch Strukturierung des Entwicklungsprozess und Arbeitsvorbereitung aus. Er ist organisationslastig und dokumentenzentriert. eXtreme Programming hingegen ist ein leichtgewichtiger Prozess, da hier der Fokus auf die Implementierung gerichtet ist. Auf umfangreiche Dokumentation und Organisation wird in XP weitgehend verzichtet.

6. Quellenverzeichnis

[1] B. Boehm, “Software Engineering”, *IEEE Transactions on Computers*, C-25(12): pages 1226-1241, December 1976

[2] H. Balzert, „Lehrbuch der Softwaretechnik“, *Spektrum Verlag, Heidelberg – Berlin*, 1998

[3] P. Naur, B. Randell,. “Software Engineering: Report of a conference sponsored by the NATO Science Committee”, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO (1969) 231pp.

[4] P. Jalote. “An integrated approach to software engineering” 2nd Edt., *Springer-Verlag New York, Inc.*, 1997.

[5] I. Sommerville, “Software Engineering”, 8th Edt., Pearson Education, *München*, 2007

[6] J. Ludewig., H. Lichter , Software-Engineering, dpunkt.verlag, *Heidelberg*, 2007

[7] The Standish Group International, Incorporated, „The Chaos Report (1994)“, *Boston, Massachusetts*, www.standishgroup.com/sample_research/chaos_1994_1.php , Letzter Zugriff: 15.05.2007

[8] S. Ueberhorst, „Wer zu spät testet, verschleudert Geld“, www.tecchannel.de, 2006, http://www.tecchannel.de/entwicklung/448205/ , Letzter Zugriff: 15.05.2007

[9] B. Boehm, “A View of 20th and 21st Century Software Engineering”, 28th ICSE, may 20-28, 2006 *Shanghai, China*.

[10] B. Boehm: “A Spiral Model of Software Development” and Enhancement. *IEEE Computer*, Vol.21, Ausg. 5, Mai 1988, pp 61-72

[11] P. Kruchten: The Rational Unified Process - An Introduction, Addison-Wesley, 1999

[12] I. Jacobson, G. Booch, J. Rumbaugh: “The Unified Software Development Process“, Addison-Wesley, 1999

[13] Rational Software, „The Rational Unified Process: Best Practices for Software Development Teams“, Rational Software Corporation, 1998

[14] R. Pressman, “Software Engineering. A Practitioner’s Approach”, 6th Edt., Mcgraw-Hill Higher Education, *New York*, 2005

[15] C. Bunse, A. von Knethen, “Vorgehensmodelle kompakt”, SpektrumVerlag, Heidelberg, 2002

[16] R. Martin, “Agile Software Development, Priciples, Patterns, and Practices”, Pearson Education, *New Jersey*, 2003

[17] K. Beck, “Extreme Programming - Das Manifest” Addison Wesley, *München*, 2003.

[18] C. Dogs, T. Klimmer, “Agile Software-Entwicklung kompakt”, mitp-Verlag, *Bonn*, 2005.

[19] A. Cockburn, L. Williams. “The costs and benefits of pair programming”, pages 223–243, Addison-Wesley Longman Publishing Co., Inc., *Boston, MA, USA*, 2001.

[20] A. Lux, „Angewandte Prozessmodelle“, *Javamagazin*, Vol. 12, 2002, pp. 73 – 76

[21] B. Boehm, „Project termination doesn’t equal project failure“, *IEEE Computer*, September 2000

Softwareprozesse sind auch Software –

Aspekte einer Entwicklung

Manfred Jürgen Primus
0060243
mprimus@edu.uni-klu.ac.at

Abstract

1987 hat Leon Osterweil bei der ICSE den Artikel „Software Processes are Software too“ veröffentlicht. 10 Jahre später wurde dieser Artikel zum einflussreichsten Artikel der 9. International Conference on Software Engineering (ICSE) gekürt. Dieser Artikel greift die Aussage L. Osterweils aus dem Jahre 1987 auf und untersucht die zwei im Abstand von zehn Jahren im Rahmen der ICSE veröffentlichten Publikationen genauso wie die kontrovers dazu verfassten Ansichten M. Lehmans.

Wurde der Ansatz Osterweils weiterverfolgt?

Diesbezüglich sind einige Entwicklungen erwähnenswert. Er selbst hat mit APPL/A und Little-JIL zwei Entwicklungen mitgetragen, die seinen Ansatz untermauern sollten. Hierbei handelt es sich um zwei Softwareprozessprogrammiersprachen, die sehr unterschiedlich sind. Die eine ist an die Programmiersprache Ada angelehnt, die andere – Little-JIL – ist eine grafische Sprache. Hier will ich die grundlegenden Ideen aufzeigen und damit die Unterschiede hervorheben.

Die dritte Programmiersprache für Software Prozesse, auf die ich näher eingehen will, ist der heute sehr aktuelle und gebräuchliche Rational Unified Process (RUP). Philippe Kruchten hat in Bezug auf den Rational Unified Process gesagt [Kruchten 03], dass mittels RUP der Softwareentwicklungsprozess als Software behandelt werden kann. Ein interessanter Aspekt hierbei ist, dass RUP selber sowohl Softwareprozess als auch Software ist.

1. Einleitung

Die Aussage, das Softwareprozesse auch Software seien sollen, hat mich zuerst sehr überrascht. Es erschien mir vorerst fast paradox und erinnerte mich an die Frage, was zuerst existierte - die Henne oder das Ei. Wenn man Osterweils ersten Artikel „Software Processes are Software too“ aus dem Jahr 1987 liest, merkt man, dass die Begriffe aus dem Titel erst einmal genau definiert werden müssen, um die von ihm vorgestellte Grundidee zu verstehen.

Deshalb sollen im zweiten Kapitel die wichtigsten Aussagen aus dem Aufsatz aus dem Jahr 1987 wiedergegeben werden. Es war sicher ein neuer Ansatz in der jungen wissenschaftlichen Disziplin den Softwareprozess auch als Software zu sehen und dürfte Grund für einige Diskussionen gegeben haben. Was der Grund war, warum dieser Artikel zum einflussreichsten der 9. ICSE gewählt wurde, habe ich nicht herausgefunden. Nachdem Leon Osterweil im Aufsatz „Software Processes are Software too, Revisited“ einige Aussagen aus dem Leitartikel untermauert und einige Aspekte hinzugefügt hat, fließt auch dieser in meinen Artikel mit ein.

Wie schon erwähnt, hat es um die Aussagen Osterweils einige Diskussionen gegeben. Ein Kritiker war M. M. Lehman. Er geht mit Osterweil nicht konform und gibt einige stichhaltige Gründe an, warum die Idee von Leon Osterweil nicht oder nur bedingt funktionieren kann. Lehmans Artikel waren unter anderem auch ein Grund dafür, warum auch ich bezweifelte, dass eine Umsetzung der Forderungen und Ideen von Osterweils Arbeiten praktikabel realisiert worden ist. Die Meinung von M. Lehman will ich im dritten Kapitel aufzeigen.

Nachdem sich mir die Frage aufdrängte, ob es nun eine von Osterweil geforderte Softwareprozess-Programmiersprache gibt, die heute in Verwendung ist, oder eine andere Verwirklichung seiner Idee, machte ich mich auf die Suche nach Prototypen oder heutefür die Softwareprozess-Programmierung verwendeten Programmiersprache.

Mit APPL/A fand ich eine auf Ada basierende Programmiersprache für den Softwareentwicklungs-

prozess. Kontrastierend dazu möchte ich im Kapitel fünf Little-JIL vorstellen, eine grafische Programmiersprache, bei der man sich vorstellen könnte, dass sie in Verwendung ist, wobei man aber keinen Hinweis dafür findet. Sie ist aber immer noch in Entwicklung.

Wohl in Verwendung ist der Rational Unified Process. Die Kernideen und den Prozess selbst möchte ich im sechsten Kapitel darstellen.

2. Software Processes are Software too – zweiter Rückblick

Im 1987 erschienenen Paper erklärt Leon Osterweil den Begriff Prozess und Prozessbeschreibung. Er verwendet hierzu ein Beispiel aus der „realen Welt“, das Kochen nach einem Kochrezept [Osterweil 87]. Wenn man nach einem Rezept kocht, ist das Kochen der Prozess und das Kochrezept die Prozessbeschreibung. Wobei zu beachten ist, dass das Rezept eine starre Beschreibung ist, aber die Ausführung – wer selber kocht weiß dies – sehr dynamisch sein kann.

Auch in der Software ist der Prozess die dynamische Entität einer Prozessbeschreibung. Diese Prozessbeschreibung ist für Menschen zwar leicht lesbar, kann aber mitunter zu Problemen führen. Dijkstra [Dijkstra 68] hat dies 1968 anhand der GOTO-Anweisung beschrieben.

Die Erstellung von Software wird ebenso wie das Herstellen und Entwickeln von Autos, Möbeln oder ähnlichem als ein reales Produkt gesehen. Es gibt hier zwar Parallelen, aber Software bleibt trotzdem ein nicht greifbares, nicht materielles, unsichtbares Produkt. Osterweil schreibt auch, dass es keine konkrete Prozessbeschreibung zum Realisieren von Softwareprozessen gibt.

Aus was besteht nun eigentlich Software? Wenn es sich um ein großes Projekt handelt, klarerweise aus tausenden Zeilen von Code. Aber man produziert auch Unmengen an Begleitmaterial, die gesamten Dokumentationen, die Informationen bezüglich der Anforderungen und des Design, die Spezifikationen und Testfälle und die Benutzer- und Wartungshandbücher um nur die wichtigsten zu nennen. Osterweil findet es erstaunlich, dass wir, obwohl es keine präzise Softwareentwicklungsprozess-Beschreibung gibt, trotzdem gute Produkte entwickeln. Welche Software könnten wir erst entwickeln, wenn wir die bis dahin eingesetzten Prozesse zur Softwareentwicklung klar definieren könnten? Wenn wir eine genaue Beschreibung dieses Softwareprozesses hätten? Wenn wir uns den Computer als Hilfswerkzeug in größerem Maß zu Nutze machen könnten?

Eine Lösung, die Osterweil aufgreift, ist die Verwendung zeitgemäßer Programmiertechniken und Formalismen zum Beschreiben des Softwareentwicklungsprozesses.

Man könnte jetzt einwerfen, dass es schon Lösungsansätze gibt, wie das Wasserfallmodel, das Spiralmodell oder das V-Modell. Das sind aber nicht Lösungen, wie sich Osterweil das vorstellt. Er schreibt sogar, dass beim Versuch das Wasserfallmodel als Basis für eine Nutzung als Prozessprogramm zu verwenden zu trivialem und unvollständigem Code führt. Nach seiner Vorstellung sollten wir den Softwareprozess mehr programmieren, so als ob wir ein Programm schreiben. Denn beide, also das Softwareprozess-Programm und ein herkömmliches Programm sind Prozessbeschreibungen, dienen dazu viele, aber auch komplexe Informationen zu transformieren, modifizieren oder zu erzeugen. Mit Hilfe des Softwareprozess-Programmes sollte man grob gesprochen alle Daten sammeln, diese durch Bearbeitung der Werte und Daten in Software-Objekte umwandeln und durch das Verbinden dieser Objekte wiederum ein fertiges Softwareprodukt herausbekommen. Bei einer Applikation können die Daten klein und überschaubar, aber auch groß und komplex sein. Bei einem Softwareprozessprogramm werden die einzugebenden Daten vorwiegend komplexer Natur sein. Aber in beiden Fällen geht es laut Osterweil um das gleiche Ziel.

Bei Applikationen ist die Lösung des Problems – also die Beschreibung des Prozesses – der ausführbare Code. Betrachtet man die Forderung Osterweils, dann ist, wenn man den Softwareprozess als Programm sieht, die Problemlösung das Prozess-Programm selbst.

Wenn man nun ein Prozess-Programm als Hilfestellung beim Softwareentwicklungsprozess hätte, wäre ein großer Vorteil, die Daten in konstanter Form gespeichert zu haben. Die Daten wären dann, wie es im Jahr 1987 war, nicht an eine oder eine Handvoll Personen gebunden sondern auf Grund des Softwareprozess-Programms wären sämtliche anfallenden Daten für alle Beteiligten verfügbar. Hätte man eine Programmiersprache für den Softwareprozess, hätte man einen weiteren Vorteil. Durch diese Sprache ließen sich die Daten einheitlich speichern, sie könnten in vielen Varianten sichtbar gemacht werden. Dadurch hätte man unter anderem auch die Möglichkeit für Kunden lesbare und verständliche Abstraktionen verfügbar zu haben. Die Beteiligten am Softwareprozess verfügten über ein Werkzeug, mit dem sie auf verschiedenen Abstraktionsebenen über das zu erstellende Produkt diskutieren und reden könnten.

Durch das Definieren einer Prozessprogrammiersprache könnte man die eingegebenen Daten automatisch analysieren, aufbereiten und interpretieren lassen.

Es wäre ein großer Schritt in der Weiterentwicklung des Softwareprozesses, wenn man es wirklich schaffen würde, sie durch eine Softwareprozess-Programmiersprache zu materialisieren.

Zehn Jahre später wurde sein Aufsatz aus dem Jahre 1987 zum einflussreichsten Paper der „9. Internationale Conference on Software Engineering" gewählt. In diesem Paper unterstrich Osterweil einige Aussagen, die er zehn Jahre vorher veröffentlicht hat. Vor allem unterstreicht er die Forderung nach der Erstellung einer Programmiersprache zum Programmieren des Softwareentwicklungsprozesses. Hier erwähnt er die Sprache JIL, die in einem weiteren Kapitel noch behandelt wird.

In dieser Abhandlung [Osterweil 97] hebt er z. B. hervor, dass man dazu übergegangen ist, verschiedene Werkzeuge, die man als Modelle von Software benutzt, auch im Softwareentwicklungsprozess zu nutzen. Beispiele sind Petri Netze, Datenflussdiagramme, Endliche Automaten u. a. Hier muss man aber die Grenzen der einzelnen Modelle beachten. Jedes Modell legt sein Hauptaugenmerk auf bestimmte Bereiche. Zur Veranschaulichung von Parallelität werden oft grafische Notationen benutzt, aber die Ursachen der Parallelität können mit der zumeist unzureichenden Semantik dieser Modelle nicht oder nur schwer veranschaulicht werden. Dieses Beispiel ist besonders im Softwareentwicklungsprozess beachtenswert, da hier sehr viele Dinge parallel ablaufen.

3. Warum ein Softwareentwicklungsprozess nur bedingt als Software behandelt werden kann

Bei beiden Veröffentlichungen von „Software Processes are Software too" tat M. M. Lehman seinen Standpunkt zu dem von Osterweil beschriebenen Softwareentwicklungsansatz kund.

Er findet den Ansatz gut und nützlich. Er kann sich ein Funktionieren aber nur vorstellen, wenn ein umfangreiches Modell der Domäne oder des Systems, das Bestandteil des Problems ist, bekannt ist und verstanden wird. Weiters müssten Strategien und Algorithmen von vorn herein bekannt sein. Ebenso wie benötigte Ressourcen. Andernfalls ist der Ansatz von Osterweil für ihn nutzlos.

Weiters kritisiert er ein Beispiel, das Osterweil in seiner Publikation veröffentlicht hat. Lehman schreibt [Lehman 87]:

> *„Expressions like '**code wrong: change_code**' or '**create_design**' in the body of a Process Program do nothing to clarify the process"*

Also ist die Tätigkeit entweder trivial oder der Versuch die Illusion eines Fortschrittes zu erwecken.

Als Beispiel, in dem ein Prozessprogramm funktionieren könnte, nannte er Domänen, in denen das benötigte Wissen für die Softwareentwicklung komplett vorhanden ist. So eine Domäne wäre ein mathematisches Problem, weil für diese Applikationen alles in mathematischen Termen ausgedrückt werden könnte. Früher wurden diese Programme ad hoc programmiert.

Lehman findet den Ansatz von Osterweil also nur sinnvoll in gewissen abgegrenzten Bereichen.

Auch D. Notkin [Notkin 88] sagt, dass kein interessantes Softwareprojekt a priori definiert werden kann, weil das Projekt von Kreativität genauso wie von der Analyse von früheren Schritten im Prozess beeinflusst ist. Also wäre es nicht möglich ein Prozessprogramm zu konstruieren: Entweder sind die Funktionen des Programms leicht zu implementieren oder aber gar nicht.

4. APPL / A: Prototyp einer Softwareprozess-Programmiersprache

Trotz der wohl auch begründeten Gegenmeinungen gab es Bemühungen und Forschungen um die Idee Osterweils umsetzen zu können. APPL / A ist eine Programmiersprache um Softwareentwicklungsprogramme beschreiben und schreiben zu können und basiert auf der Programmiersprache ADA. APPL / A wurde als logische Folge der Arbeit von Leon Osterweil aus dem Jahre 1987 auf Grund der Forderung nach einer Programmiersprache zum Schreiben von Programmen als Unterstützung im Softwareentwicklungsprozess entwickelt, wobei Leon Osterweil federführend beteiligt war.

Das Ziel der Entwickler von APPL / A war einen Prototyp für eine Programmiersprache zum Schreiben von Programmen für Softwareentwicklungsprozesse zu schaffen. Als Ziele setzten sie sich [Sutton 95]:

- Ausführbarkeit: um den Prozess automatisiert ablaufen lassen zu können
- Möglichkeit zur Kontrolle und zur Definition von Datentypen: damit Prozessaktivitäten und -produkte dargestellt werden können; zum Programmieren von prozessunabhängigen Details
- Abstraktion, Kapselung und Modularität
- Darstellung von Abhängigkeiten: um Abhängigkeiten zwischen den einzelnen Prozessschritten und Akteuren aufzuzeigen

- Datenspeicherung und -austausch
- Programmierbarkeit von Anwendungen: zur Kontrolle und Datenabstraktion, insbesonders zum Erstellen von Alternativen
- Nebenläufige und rückwirkende Kontrolle
- Prozesszugehörige Daten: Möglichkeit Daten, die im Laufe des Prozesses erstellt oder erhalten werden, unterstützt in den Prozess einfließen zu lassen
- Management für flexible Transaktionen: um Daten und Ressourcen zwischen den einzelnen Benutzern zu verteilen, ohne Inkonsistenzen zu verursachen.

Dazu mussten der Sprache Ada vier wichtige Erweiterungen hinzugefügt werden.

Die erste betrifft „Relation Units“, die wie eine mathematische Notation die Darstellung der Beziehung zwischen den einzelnen Einheiten im Softwareentwicklungsprozess darstellen. Man benötigte hier auch eine allgemein gültige, aber flexible Datenstruktur um die anfallenden „Produkte“ und Daten darstellen und speichern zu können. Nachdem auf Daten nicht nur von einer Stelle aus zugegriffen werden soll, muss sichergestellt werden, dass die Daten persistent bleiben.

Als nächstes wurden „Trigger units“ eingeführt. Trigger units sind Kontrolleinheiten, die auf Ereignisse, die die Relation units betreffen, reagieren und bei gleichzeitigem Zugriff auf Einträge auch diese Nebenläufigkeit kontrollieren. Trigger units sorgen ebenfalls dafür, dass Nachrichten von einer Einheit zur anderen transportiert werden, wenn sich bei relevanten Daten etwas ändert. Weiters sind sie für Berechnungen, für das Führen der Protokolle, generell für sämtliche Aktionen, die die Relation units betreffen, zuständig.

Die dritte Erweiterung sind „Predicate units“. Diese erlauben die Definition von Bedingungen bei Beziehungen. Dabei kann es sich um bestimmte Merkmale, wie um referentielle Integrität oder um Eindeutigkeit handeln.

Die vierte Erweiterung behandelt das Management der Transaktionen. Die in APPL/A verwendeten Transaktionsbefehle sind die serial, atomic, suspend, enforce und allow Anweisungen. Diese Anweisungen geben an, wie mit den Daten bei Transaktionen verfahren werden soll bzw. kann.

Wenn man sich Beispiele in [Sutton 95] ansieht, denkt man wieder an die Aussage Lehmans, dass manche Codeabschnitte wiederum nur triviale Tätigkeiten beschreiben. APPL / A wurde entwickelt, um Versuche mit einer Prozessbeschreibungssprache beim Entwickeln von Prozessprogrammen machen zu können. Mit den Konstrukten von APPL / A konnten sie einige unterschiedliche Softwareprozesse programmieren, einschließlich der Aufgaben im Softwareentwicklungsprozess- und im Management-bereich.

Wenn man sich die unter Ziele definierten Punkte ansieht, merkt man, wie weit gestreut der Aufgabenbereich im Softwareentwicklungsprozess ist. Durch das Experimentieren mit APPL / A konnten einige wichtige Aspekte der Programmierung eines Prozessprogrammes verdeutlicht werden. Vor allem waren dies Prozess-, Produkt- und Projektmanagement, mittel- und langfristige Aktivitäten, das Einbinden von Dokumentationen und automatisierten Tätigkeiten. Weiters waren dies das Design der Visualisierung, Aufzeichnung und Messung von Prozessdaten, die Einbindung von Änderungen im Prozess und die Wichtigkeit der Anforderungsanalyse und anderer Tätigkeiten des Softwarelifecycles.

5. Little-JIL/Juliette: Prozessdefinitions-sprache und Interpreter

Little-JIL ist eine Programmiersprache mit einer grafischen Syntax die unter Mitwirkung von Leon Osterweil entwickelt wurde. Trotzdem Little-JIL grafisch ist, hat sie eine sehr strikte präzise Semantik, fast vergleichbar mit der einer textuellen Programmiersprache.

[Sutton 97] Bei der Entwicklung von Little-JIL wurde besonders auf vier Punkte geachtet: Einfachheit, Ausdrucksstärke, Präzision und Flexibilität. Nachdem Little-JIL auf einer grafischen Syntax basiert und dadurch leicht zu benutzen und zu verstehen ist, ist sie für die Entwickler genauso wie auch für Laien, wie z. B. Kunden einfach zu interpretieren und zu handhaben. Obwohl die Sprache sehr einfach ist, ist sie dennoch ausdrucksstark. Auf Grund der verschiedenen Abläufe in einem Softwareprozess aber auch der Koordination dieser, muss die Prozessbeschreibungssprache fähig sein, diese Abläufe klar darzustellen. Die Prozessbeschreibungssprache dient im Endeffekt dazu, einen Code zu produzieren, der auch ausführbar ist. Weiters soll das Ergebnis auch analysierbar sein, um z. B. Fehler aufdecken zu können oder um eine Verlässlichkeit zu garantieren. Dazu dient die präzise Definition der Sprache. Um nachträgliche Änderungen ohne großen Aufwand durchführen zu können, muss die Sprache Flexibilität unterstützen. Flexibel sollte die Sprache auch dann sein, wenn es mehrere Möglichkeiten zum Lösen eines Problems gibt.

Mit Little-JIL wird der Prozess als Hierarchie von Steps gesehen. Das Programm ist quasi ein Baum mit Blättern, wobei jedes Blatt einen Agenten darstellt. Ein Agent kann entweder ein Mensch sein (der Reisende in

Abbildung 1 oder auch der Requirements Engineer bei der Bedarfsanalyse) sein oder eine Maschine (wie der Reservierungsautomat in Abbildung 1 oder ein im Softwareentwicklungsprozess verwendetes Werkzeug). Ein Agent bewältigt einen kleinen Arbeitsabschnitt im Softwareentwicklungsprozess und gibt entweder einen gefunden Fehler oder den erfolgreichen Abschluss seines Arbeitsschrittes zurück.

In Little-JIL wurden auch Ausnahmen berücksichtigt. Es können Ressourcen fehlen oder Parameter nicht stimmen und dazu können geeignete Exceptions entworfen werden, die von Handlern behandelt werden und dem Eltern-„Step“ übergeben werden.

Um eine Kommunikation zwischen den einzelnen „Steps“ zu ermöglichen, können Parameter übergeben werden, wie man im Beispiel zwischen „Step“ PlanTrip und PlaneReservation sieht. Hier werden TripDates und Budget an PlaneReservation übergeben und PlanTrip erhält als Paramter TripDates und die Airline zurück.

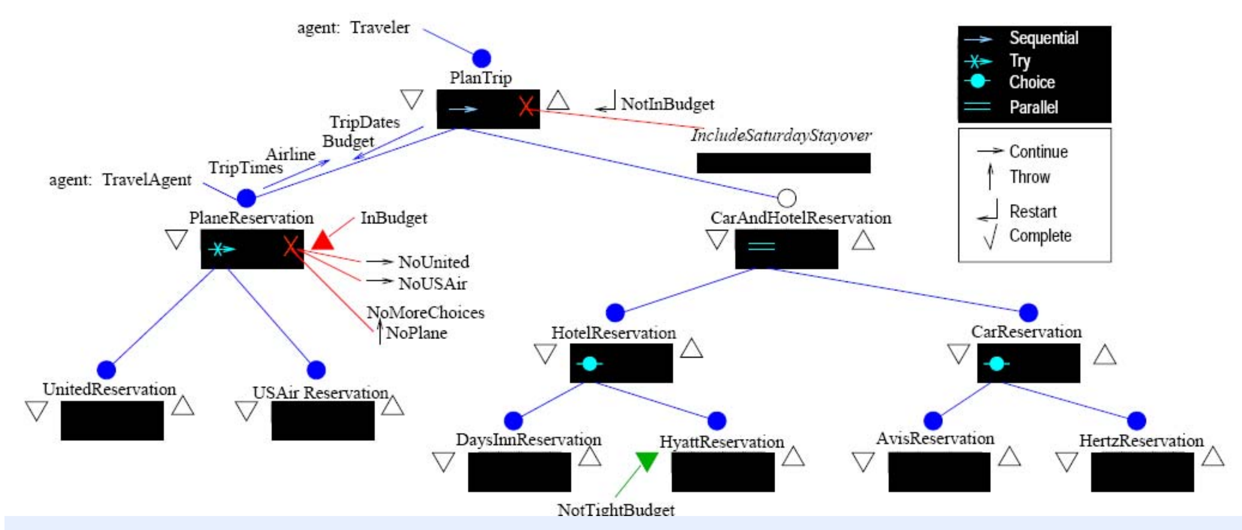


Abbildung 1: „Steps“ und Agenten in einem Reservierungsprozess Quelle [Cass 00]

Um die Funktionsweise von Little-JIL möglichst einfach erklären zu können dient Abbildung 1 [Cass 00]. Ein „Step“ repräsentiert die Übergabe eines Arbeitsschrittes mit allen zugehörigen Attributen an einen Agenten. Dem Agenten werden dabei auch Informationen und Ressourcen übermittelt (z. B. die vom vorigen Agenten erstellten Dokumente). Es müssen aber auch alle Tätigkeiten, von dem der Arbeitsschritt des Agenten abhängt, zuvor erfüllt worden sein. Für die Flusskontrolle wird definiert ob dieser „Step“ sequentiell, parallel, als Versuch oder alternativ ausgeführt wird. Nicht-Blätter können in einen oder mehrere Unterschritte weitergeführt werden, wobei der Ausführungsbeginn vom Sequenzzeichen abhängt. Im obigen Beispiel ist das Ausführen der Steps HotelReservation und CarReservation unabhängig voneinander, während die Nachfolgeschritte von PlaneReservation auf Grund des Zeichens „try“ in der Reihenfolge von links nach rechts ausgeführt werden.

Es wurde ein Mechanismus (Requisites) eingeführt, der das definieren von Bedingungen erlaubt, wann ein „Step“ beginnen darf und wann er beendet ist. Eine Bedingung kann die Beendigung eines anderen Schrittes sein oder z. B. eine Gewichtsbeschränkung für Gepäck beim Einchecken in einen Flug.

Genauso können Ressourcen in einen „Step“ einfließen. Ressourcen können wiederum Agenten (menschliche oder mechanische) sein, Werkzeuge oder andere benötigte Ressourcen.

Little-JIL wurde bzw. wird an der Universität Massachusetts entwickelt. Bei meinen Recherchen fand ich keinen Hinweis auf Erfahrungsberichte in realen Softwareprojekten mit Little-JIL.

6. Rational Unified Process

Im letzten Kapitel möchte ich nun auf den Rational Unified Process eingehen. RUP wurde bei Rational Software entwickelt, wobei der Ansatz dazu schon in den achtziger Jahren entwickelt wurde. Die erste Version von RUP wurde 1998 herausgegeben. Der für die Entwicklung Hauptverantwortliche war Philippe Kruchten.

In seinem Buch sagt Kruchten selbst, dass der Rational Unified Process wie Software entworfen,

entwickelt, vertrieben und gewartet wird. Es werden regelmäßig Updates herausgegeben, die Vermarktung erfolgt mittels Internet, es kann an die Anforderungen der Benutzer angepasst werden und es sind viele Softwareentwicklungswerkzeuge in RUP integriert.

Bei der Entwicklung von Software haben sich einige Grundsätze herauskristallisiert, die von namhaften Softwareentwicklern und Softwarefirmen propagiert werden. [Kruchten 03]:

- Iterative Entwicklung von Software: Da sich Durch den iterativen Ansatz können Änderungen in den Anforderungen bei jeder Iteration neu einfließen und verursachen weniger Probleme. Es muss nicht am Beginn der Entwicklung einer Software alles entschieden werden, sondern es passiert kontinuierlich. Dadurch ist auch Reuse leichter. Fehler können eher und leichter behoben werden, als in einer einzigen Testphase fast am Ende der Entwicklung.
- Verwaltung der Anforderungen: Dies umfasst das Finden, Organisieren, Besprechen und Verwalten der Anforderungen. Wird dies effektiv unterstützt, erhält man reduzierte Kosten, verbesserte Qualität und bessere Kommunikation unter den Beteiligten.
- Benutzung von auf Komponenten basierenden Architekturen: Auch hier spielt wieder der iterative Ansatz hinein. Dadurch kann man während des gesamten Entwicklungsprozesses entscheiden welche Komponenten man selber entwickelt oder ob eine vorhandene wiederverwendet wird. Beim Unified Prozess werden die einzelnen Komponenten erst alleine getestet und dann noch einmal nach dem Einfügen ins Ganze.
- Grafische Modellierung von Software: Im RUP wird zur grafischen Darstellung UML sehr gerne verwendet. Modelle helfen uns durch Abstraktion ein Problem und seine Lösung besser zu verstehen und einzugrenzen.
- Management der Struktur und von Änderungen: Die während der Softwareentwicklung erstellten Produkte müssen verwaltet werden. Diese Produkte werden immer wieder verändert. Diese Änderungen müssen verwaltet und dokumentiert werden. Und man muss wissen, von wem sie gerade benötigt oder durchgeführt werden.

Weitere Schlüsselfunktionen des RUP:

- Use-Case getriebene Entwicklung: In der Objekt-orientierten Softwareentwicklung sind Use-Cases weniger gebräuchlich, jedoch sind sie eine wichtige Brücke zwischen Anforderung und Test oder Design. Im RUP sind Use-Cases die Basis für den Rest des Entwicklungsprozesses
- Prozess Konfiguration: RUP ist ein Prozess-Framework; man sollte, bevor man ein Projekt beginnt, RUP so anpassen, das man in kürzest möglicher Zeit eine höchst mögliche Qualität bekommt.

Wie läuft nun der RUP ab?

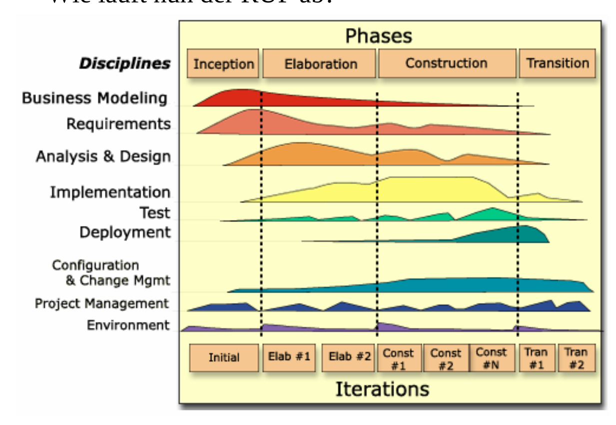


Abbildung 2: Prozessstruktur Quelle [IBM 01]

Der Ablauf einer Iteration wird in Abbildung 2 gezeigt. Der horizontale Ablauf repräsentiert die Zeit und zeigt die einzelnen Abschnitte einer Iteration. Vertikal sieht man die einzelnen Tätigkeitsbereiche mit der Wichtigkeit zum jeweiligen Zeitpunkt im Ablauf einer Iteration.

Eine Iteration ist also in vier Phasen unterteilt und nach jeder Phase wird ein Meilenstein gesetzt. Dieser wird für die Entscheidung genutzt, ob die Iteration so fortgesetzt werden kann, ob das Projekt abgebrochen werden sollte oder ob etwas geändert werden muss. Hier nun eine Auflistung der vier Phasen mit den zugehörigen Milestones:

- Inception (Anfang): Hier werden das Ziel und die Grenzen definiert. Begrenzt wird diese Phase durch den Lifecycle Objective Milestone.
- Elaboration (Ausarbeitung): Hier werden die benötigten Ressourcen ermittelt, die nötigen Aktivitäten geplant, die Funktionen festgelegt und die Architektur entworfen. Abschließend ist der Lifecycle Architecture Milestone.
- Construction (Konstruktion): Hier werden die Komponenten fertiggestellt. Der Meilenstein ist der Initial Operational Capability Milestone.
- Transition (Übergabe): Hier bekommt der Auftraggeber restriktive der Softwarebenutzer das Produkt. Inkludiert sind das Installieren, die Schulung, der Support und die Wartung des

Produktes. Der Meilenstein Product Release schließt die Phase und die Iteration ab.

Die erste Iteration wird auch als Initial Development Cycle bezeichnet, die nächsten Iterationen sind Evolution Cycles.

In den unterschiedlichen Phasen werden unterschiedliche Schwerpunkte gesetzt. In der Abbildung sieht man den Schwerpunkt in der ersten Phase, der Inception, besonders in der Anforderungsanalyse. In der zweiten Phase sieht man zusätzlich noch den Fokus bei Design und bei der Implementierung eines evtl. Prototyps. In der Konstruktionsphase wird der Fokus auf die Fertigstellung der Applikation gelegt, wobei aber im ersten Teil auch das Design noch eine größere Rolle spielt. In der letzten Phase geht der Schwerpunkt dann auf die Übergabe über, wobei hier die Konfiguration beim Kunden überwiegt.

Ein Prozess beschreibt wer was wann wie tut. Im RUP gibt es fünf primäre Elemente:

- Rollen: sind das wer
- Aktivitäten: sind das wie
- Artefakte: sind das was
- Workflows: sind das wann
- Disziplin: der Container, in der sich die vorherigen Punkte befinden

Ein Designer (Rolle) führt eine Use-Case-Analyse durch und erstellt dann ein Use-Case-Design (Aktivitäten) und erhält dann eine Realisierung eines Use-Cases (Artefakt). Die Reihenfolge, wie der Designer das macht und welche Optionen er hat, können in einem Workflow dargestellt werden.

Im Mittelpunkt des RUP steht eine Rolle. Eine Rolle beschreibt das Verhalten und die Verantwortlichkeiten eines Einzelnen oder einer Gruppe. Das Verhalten ist in Aktivitäten, die die Rolle ausführen kann, festgelegt. Die Verantwortung wird gegenüber eines oder mehrerer Artefakte ausgedrückt, welche von der Rolle erzeugt, modifiziert oder kontrolliert werden können. Rollen sind auch nicht dauerhaft an eine Person gebunden, sondern können bei Bedarf und bei Qualifikation vergeben werden. Man kann z. B. für einen bestimmten Zeitraum Designer sein und danach Reviewer mit mehreren anderen und sich dann wieder in einer anderen Rolle wiederfinden.

Aktivitäten werden durch Rollen ausgeführt und müssen ein sinnvolles Ergebnis für das Projekt liefern. Die Aktivität eine Iteration zu Planen gehört zur Rolle des Projekt Managers, das Finden von Use-Cases und Akteuren gehört zur Rolle des System Analysten und der Performance Tester ist verantwortlich für das Ausführen eines Performance Tests.

Aktivitäten werden wiederum in drei Schritte unterteilt: das Denken, das Durchführen und das Überprüfen der Aktivität.

Aktivitäten können In- und Output Artefakte haben. Artefakte sind Produkte (Informationen) die erzeugt, modifiziert oder von Aktivitäten gebraucht werden. Artefakte können Modelle, Dokumente, Source Code oder ausführbare Programme(-teile), wie z. B. Werkzeuge sein.

Diszipline sind die in Abbildung 2 gezeigten „Container", wie z. B. Business Modelling oder Requirements.

Was sind nun die Vorteile eines iterativen Prozesses gegenüber z. B. Beispiel des traditionellen Ansatzes eines Wasserfallmodels?

Das Risiko einen Fehler mitzuschleppen wird vermindert. Man erkennt Fehler oft erst bei der Übergabe des Produktes an den Kunden. Durch den iterativen Prozess können diese Fehler früher erkannt werden. Je früher ein Fehler erkannt wird, desto geringer sind die Kosten diesen zu korrigieren. Wenn ein Projekt abgebrochen werden muss, ist der Verlust ebenfalls geringer, wenn so früh als möglich abgebrochen werden kann. Das Wasserfallmodel zielt auf eine (End-)Übergabe ab, bei RUP hat man normalerweise mehrere Produktübergaben.

Änderungen können früher abgefangen werden. Die häufigsten Änderungen betreffen Anforderungen. Dies führte bei anderen sequentiellen Modellen oft zu großen Problemen. Nachdem der Auftraggeber beim iterativen Prozess schon eine frühe Version des Produktes in Händen hält, sind hier die Probleme geringer und die Chance auf ein bestmögliches Produkt ist weitaus größer.

Reuse ist leichter möglich. Bei RUP besteht die Möglichkeit auf Grund von Architekturzentriertheit und der Benutzung von Komponenten leichter Reuse zu betreiben.

Dadurch, dass das Produkt durch die Iterationen unzählige Male getestet worden ist, das die Anforderungen immer wieder nachgebessert und verfeinert wurden, verbessert sich auch die Qualität.

Im Rational Unified Process gibt es 30 Rollen die in fünf Hauptkategorien eingeordnet sind. Analysten, Entwickler, Manger, Tester, Produktion und Support. Rollen spezifizieren, wie gesagt, nicht Personen sondern Fähigkeiten und Kompetenzbereiche. Jede dieser Kategorien wird noch einmal unterteilt in spezifischen Rollen. Aus jeder Hauptkategorie soll nun ein Vertreter beschrieben werden:

- Anforderungsanalyst: Er spezifiziert eine oder mehrere Funktionalitäten des Systems unter Zuhilfenahme von Use-Cases.

- Software Architekt: Er trifft die Entscheidung, in welche technische Richtung das System geht, indem er auf Grund der Anforderungen, der Stakeholder eine Architektur für das System baut.
- Project Manager: Stellt Ressourcen bereit, verteilt diese, setzt Prioritäten, koordiniert Treffen zwischen Kunden und Teammitgliedern.
- Test Designer: Erstellt, plant, implementiert und evaluiert Testfälle.
- Technical Writer: Schreibt das Handbuch, Support Material, Hilfetexte, Texte zu den Releases usw.

Außerdem gibt es noch zusätzliche Rollen, wie z. B. den Reviewer, den Review Coordinator, Any Role (Any Role ist eine generische Rolle der Tätigkeiten zugewiesen werden können, die noch von keiner anderen Rolle ausgefüllt wird, aber benötigt wird) usw.

Dass RUP von verschiedenen Firmen genutzt wird, beschreibt Kruchten in [Kruchten 03].

7. Konklusion

Der teilweise philosophische Artikel von Leon Osterweil verführt geradezu zum Suchen und Nachschlagen von Definitionen. Was ist Software? Was ist ein Softwareprozess? Kann man wirklich sagen, dass ein Softwareprozess Software ist.

Wenn man die Ideen aus dem Leitartikel „Software Processes are Software Too“ hernimmt und mit Kapitel 6 Rational Unified Process vergleicht, findet man einige Ideen von Osterweil wieder. Jedoch musste für RUP keine Softwareentwicklungsprozess-Programmiersprache entwickelt werden. Die Unified Modelling Language (UML) wurde zwar parallel zu RUP entwickelt und wird bei Benutzung von RUP auch massiv eingesetzt, ist aber keine Programmiersprache, sondern eine Möglichkeit, verschiedene Sichten im Zuge des Softwareentwicklungsprozesses grafisch darzustellen.

Trotzdem könnte RUP seine Forderung nach einem umfassenden Werkzeug erfüllen, das die Softwareentwickler unterstützt und den Software-entwicklungsprozess als Software behandeln lässt.

Mit RUP hat man ein Framework an Werkzeugen in der Hand, also Software mit der man Daten bearbeiten, speichern, visualisieren, transformieren, usw. kann.

Vergleicht man RUP mit APPL / A sieht man die für APPL / A definierten Ziele fast vollständig durch RUP realisiert, obwohl RUP im Grunde keine Programmiersprache ist, sondern eine Software.

Obwohl Philippe Kruchten auf den Titel von Leon Osterweils Artikel „Software Processes are Software Too“ Bezug nimmt und sagt, RUP sei eine Realisierung der Idee von Osterweil, wurde die Wurzel der Entwicklung von RUP schon in den achtziger Jahren von Rational Systems gelegt.

Leon Osterweil hat eine rege Diskussion entfacht, die mit der Prozess-Programmierungssprache Little-JIL eine interessante grafische Realisierung unter seiner Leitung ergeben hat. Die Praktikablere – so schein mir – ist jedoch der Rational Unified Process.

8. Referenzen

[Osterweil 87] L. Osterweil, “Software Processes are Software Too”, *ACM*, 1987, pp. 2-13.

[Dikstra 68] Edsger W. Dijkstra, "Go To Statement Considered Harmful," *CACM* 11, March 1968, pp. 147-148.

[Osterweil 97] L. Osterweil, “Software Processes are Software Too, Revistied: an Invited Talk on the Most Influential paper of ICSE 9”, *ACM*, 1997, pp. 539-548.

[Notkin 88] D. Notkin, “The Relationship Between Software Development Environments and the Software Process”, *ACM*, 1988, pp. 107-109.

[Lehman 87] M. M. Lehman, “Process Models, Process Programs, Programming Support”, *ACM*, 1987, pp. 14-16.

[Sutton 95] S. M. Sutton, Jr., D. Heimbigner, L. J. Osterweil, “APPL/A: A Language for Software Process Programming”, *ACM Transactions on Software Engineering and Methodology*, Vol 4 Nr. 3, July 1995 Pages 221-286.

[Cass 00] A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, S. M. Sutton, A. Wise, “Little-JIL/Juliette: A Process Definition Language and Interpreter”, *ACM*, Limerick, Ireland, 2000, pp. 754-757.

[Sutton 97] S. M. Sutton, Jr, L. J. Osterweil, “The Design of a Next-Generaton Process Language”, , pp. 142-158.

[Kruchten 03] Ph. Kruchten, *The Rational Unified Process: An Introduction, third Edition*, Addison-Wesley, Amsterdam, December 2003.

[IBM 01] Ph. Kruchten, “What is the Rational Unified Process?”, *http://www.ibm.com/developerworks/rational/library/content/RationalEdge/jan01/WhatIstheRationalUnifiedProcessJan01.pdf*, 2001, last seen 21. Mai 2007.

Inspections Reviewed
Sommersemester 2007

Fritz Genser
9560011
fgenser@edu.uni-klu.ac.at

Abstract

Design und Code Inspektionen, im Allgemeinen Reviews genannt, haben eine lange Tradition. Bereits in den 70er Jahren entwickelt, wurden Reviews durchgeführt um Fehler im Design oder der Implementierung frühzeitig zu erkennen und zu beseitigen.

Ständige Reviews im Softwareentwicklungsprozess erhöhten die Qualität der Software und senkten gleichzeitig Kosten für spätere Fehlerbeseitigungen und Wartung.

Dieser Artikel legt den Fokus auf Code Inspektionen, wie sie seit fast dreißig Jahren – in nahezu unveränderter Form – existieren und gibt einen Überblick über aktuelle Werkzeuge und Verfahren, die Reviews erheblich erleichtern.

1. Michael Fagan

Als Vater von Code Inspektionen gilt heute der frühere IBM Mitarbeiter Michael Fagan.

Fagan, ein gelernter Physiker, beschreibt im Vorwort von *Software Inspection – An Industry Best Practice* seine Frustration während eines Software Projekts bei IBM [3]. Bereits zwei Projektleiter scheiterten daran, bis 1972 Fagan vom Hardwaresektor in die Softwaresparte wechselte. Er beschreibt die damaligen Zustände als schiere Katastrophe. Die Softwareentwicklung wurde nach dem traditionellen Life-Cycle Modell propagiert, entwickelt wurden Produkte aber meist nach dem folgenden Schema:

- Spezifiziere einige Anforderungen
- Schreibe ein wenig Code, um ein Subset von den Anforderungen zu erfüllen
- Modul-teste den Code, welcher dem Subset gerecht wird
- Integriere den Code in das System
- System-teste den Code
- Wenn der Code zufriedenstellend ist, abstrahiere das Design vom Code und schreibe es in die Form einer Designspezifikation
- Wiederhole, bis alle Anforderungen erfüllt sind, oder solange es der Zeitplan zulässt
- Schreibe die Dokumentation von den erfüllten Anforderungen
- Installiere das System beim Kunden
- Führe Fehlerbehebungen durch, die durch den Kunden aufgezeigt werden, während die Entwickler an der nächsten Version arbeiten.

Aus seiner Verzweiflung heraus führte Fagan erstmalig Methoden aus der Hardwareerzeugung ein, in der die Entwicklung von neuer Hardware ständig im Analyse-, Design- und Verlaufsprozess von neuem überprüft werden musste. Dies senkte die Fehlerrate und teure Korrekturen in der Endfertigung.

Seine Ergebnisse veröffentlichte er 1976 in seinem Paper [1], welches auch das Ausgansmaterial für diese Arbeit darstellt.

Für die Entwicklung der „*Fagan Inspection*“ und deren weltweiten Einsatz, wurde Fagan von IBM mit der höchsten Auszeichnung, dem „*Individual Corporate Achievement Award*“, geehrt.

Nach zwanzig Jahren Tätigkeit bei IBM, gründete er 1989 die Firma Michael Fagan Associates und ist seither Berater in Qualitätssicherheit und seines patentierten „*Fagan Defect-Free Process*“ [7].

2. Fagan Inspektionen

Boehm stellte die ungefähre Aufteilung der Fehlerverteilung innerhalb des Life-Cycle dar [13]. Abbildung 1 zeigt, dass bereits über 50% der Fehler eines Systems noch vor der Programmierphase auftreten. Je früher nicht entdeckte Fehler auftreten desto teurer wird die Fehlerbehebung in den nächsten Phasen. Fehler können sich auch unterschiedlich stark fortpflanzen. Dies bedeutet, sie können entweder keinen, einen oder mehrere Fehler in beliebig vielen

Folgedokumenten verursachen. In seinem Buch *Software Engineering Economics* [5], analysierte Boehm mehrere Projekte und deren Kosten für die Beseitigung von Fehlern in den unterschiedlichen Phasen der Projektzyklen (Abbildung 2).

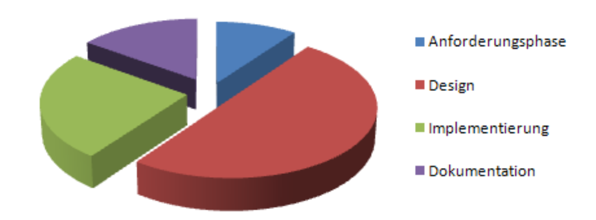


Abbildung 1: Durchschnittliche Fehler in den Lebenszyklen

Durch frühzeitige Fehlererkennung und -behebung können nicht nur Ressourcen eingespart werden, zudem erhöht sich auch die Qualität des fertigen Produkts.

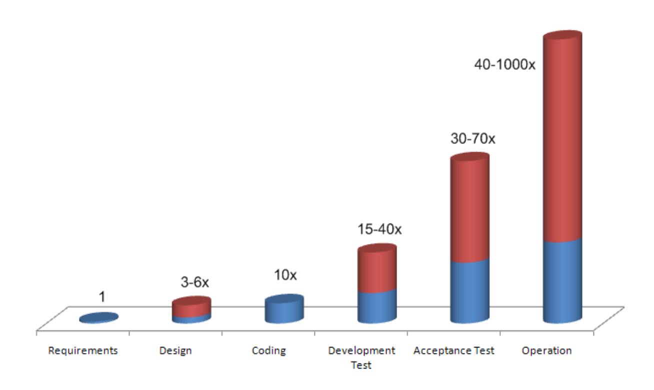


Abbildung 2: Kostenfortpflanzung eines nicht gefunden Fehlers bis zum Betrieb [5]

Eine Möglichkeit der Fehlererkennung ist die Inspektion. Fagan beschreibt eine Inspektion als eine formale, effiziente und ökonomische Methode um Fehler im Design und Code zu finden [1].

Eine formale Inspektion wird mit einem extra damit beauftragten Team zu bestimmten Zeitpunkten durchgeführt. Das zu prüfende Dokument wird dabei mittels einer Kontrollliste Punkt für Punkt durchgegangen.

Informelle Inspektionen laufen wesentlich flexibler ab, da es keine fixen Zeitpläne oder Aufgabenlisten gibt (siehe auch Kapitel 2.4). Formale Inspektionen sind normalerweise effizienter bei der Fehlersuche, informelle können öfters stattfinden.

2.1 Team

Ein Inspektionsteam besteht üblicherweise aus vier Personen: dem Moderator, Designer, Programmierer und einem Tester. Jedes Teammitglied hat das Finden von Fehlern während des Vorbereitungsprozesses und der stattfindenden Meetings, zur Aufgabe.

Der **Moderator** ist die Hauptperson bei einer erfolgreichen Inspektion. Er sollte möglichst ein erfahrener Programmierer sein, muss aber zwangsläufig nicht ein Experte im zu überprüfenden Dokument sein. Es können durchaus positive Synergien entstehen, wenn der Moderator nicht an dem Projekt beteiligt ist.

Gilb und Graham gehen noch weiter und definieren die ursprüngliche Rolle des Moderators von Fagan neu: er wird zum **Inspektionsleiter** [2]. Er muss nicht nur ein Treffen moderieren, sondern es planen und organisieren. Er ist gleichzeitig Betreuer, Lehrer und Administrator. Ein Manager, Freund und Unterstützer der restlichen Inspekteure. Die ursprünglichen Aufgaben, wie zum Beispiel die Weitergabe von Inspektionsresultaten innerhalb eines Tages, gehören selbstverständlich auch dazu.

Der **Designer** ist zuständig für das Programmdesign. Er erklärt im Allgemeinen, welcher Bereich gerade überprüft wird. In weiterer Folge werden von ihm Details zu den einzelnen Passagen erläutert – die Logik, verwendete Pfade und Abhängigkeiten. Er ist die Hauptperson bei der Designinspektion, die vor der Codierungsphase stattfindet.

Der **Programmierer** ist verantwortlich für den überprüften Code. Der Programmierer muss nicht begründen, warum er bestimmte Methoden oder Lösungen zu Problemen gewählt hat. Im Gegenteil bei der Codeinspektion geht es lediglich um das Auffinden von Fehlern in einem Dokument. Jede Zeile des Codes und jeder mögliche Zweig muss dabei erklärt werden.

Der **Tester** sollte ein Experte auf seinem Gebiet sein. Er ist für das Schreiben oder die Ausführung von Testszenarien verantwortlich beziehungsweise für das generelle Testen des Designs oder des durchzugehenden Codes.

Eine weitere Rolle ist die des **Protokollführers**. Er hat zur Aufgabe die gefunden Fehler schriftlich festzuhalten. Er muss keinerlei technische Erfahrungen mitbringen, sie wären aber von Vorteil. In kleinen Teams übernimmt der Moderator diese Rolle.

2.2 Fehlerkategorisierung

Don't call them bugs, call them 'quality improvement opportunities'.

- Graham Babel, 1990

Das Ziel jeder Inspektion sollte ganz dem Auffinden von Fehlern verschrieben sein. Diskussionen während eines Meetings sollten nur soweit geführt werden, wie sie dem Auffinden von Fehlern dienen. Diese „*defects*" werden vom Protokollführer notiert und klassifiziert. Es gibt eine Unterscheidung zwischen einfachen und schweren Fehlern. Einfache Fehler haben keine gröberen, ökonomischen Auswirkungen auf das Projekt und sind normalerweise lokal begrenzt.

Schwere Fehler haben eine dramatischere Auswirkung auf das Projekt, sollten diese nicht erkannt und entfernt werden. Sie können unter Umständen das gesamte Projekt gefährden. Diese Art von Defekt zieht hohe Kosten für die Beseitigung in vorausgegangen Dokumenten nach sich. Typische Vertreter von schweren Fehlern sind falsch implementierte Benutzeranforderungen.

Die NASA erweiterte in Ihrem *Software Formal Inspection Standard* die Kategorisierung eines Fehlers um einen Typ (Data, Requirements compliance, Interface, Logic, Standards compliance, Performance und Readability) [6]. Sie trifft damit gleichzeitig Aussagen über die Qualität des Codes, sowie Hilfestellung beim Entfernen des Mangels. Dies ist vor allem wichtig, wenn ein großer Personenkreis an dem Projekt.

Mit dieser Typisierung widerspricht die NASA Gilb und Graham, die eine breitere Klassifikation für unnötig halten. Schließlich wäre eine Diskussion – so Gilb und Graham – eine reine Zeitverschwendung, da der Fehler sowieso verbessert werden muss. Dennoch beharren viele Firmen auf eine zusätzliche Einteilung, da verschiedene Fehlergruppen, nach dem Review, von verschiedenen Spezialisten erledigt werden können.

2.3 Dauer und Follow-Up von Inspektionen

Die Dauer eines Inspektionsmeetings hängt von vielen Faktoren ab. Sie sollte jedoch nie länger als zwei Stunden andauern, da die Konzentration nach dieser Zeitspanne stark sinkt. Es gibt einige Abbruchkriterien oder Fälle in denen keine Inspektion stattfinden sollte:

- **Es werden zu viele Fehler gefunden**
  Typischerweise gibt es in Unternehmen Richtlinien, die die Anzahl der gefundenen Fehler in einer Inspektion festlegen. Die Grenze sollte hoch, aber nicht unerreichbar sein. Das Inspektionsteam ist schließlich nicht für Debugging zuständig.
- **Nicht alle Mitglieder sind erschienen**
  Jedes Teammitglied hat seine stark spezialisierte Funktion. Durch ein Fehlen können mitunter Fehler übersehen beziehungsweise Teile eines Dokuments nicht richtig wiedergegeben werden. Es ist besser ein Meeting ganz ausfallen zu lassen, bevor es nur zu Teilen stattfindet.
- **Mangelnde Vorbereitung**
  75% der gefundenen Fehler in einer Inspektion werden durch individuelle Vorbereitungen bereits gefunden [8]. Der Moderator sollte daher am Anfang jeder Sitzung die Vorbereitungszeiten der anderen Mitglieder erfragen. Er sollte dann abwägen, ob die Sitzung nicht wegen mangelnder Vorbereitung verschoben werden sollte.
- **Nicht alle Fehler der letzten Sitzung wurden verbessert**
  Es hat keinen Sinn wertvolle Ressourcen in ein Meeting zu stecken, dessen Inhalte noch nicht vollständig verbessert wurden. Inspektionen sind teuer. Die Verbesserung einer nicht fertigen Sache hat keinen Sinn.

Ein wichtiger Faktor bei der Codeinspektion ist die Anzahl der zu überprüfenden Zeilen pro Stunde. Glen Russell von Bell-Northern Research (BNR) hat mehrere Inspektionen mit demselben Code bei unterschiedlichen Lesegeschwindigkeiten von 150 LOCs (lines of code) bis 750 LOCs pro Stunde durchgeführt [14]. Bei der langsamsten Inspektionslesung wurden in etwa 37 Fehler pro tausend Zeilen Code gefunden. Laut Schätzungen von Russell sind das zirka 80% der Fehler. Hingegen konnte in der schnellsten Inspektion (750 LOCs pro Stunde) nur mehr acht Fehler oder rund 17% der Fehler auf tausend Zeilen Code entdeckt werden.

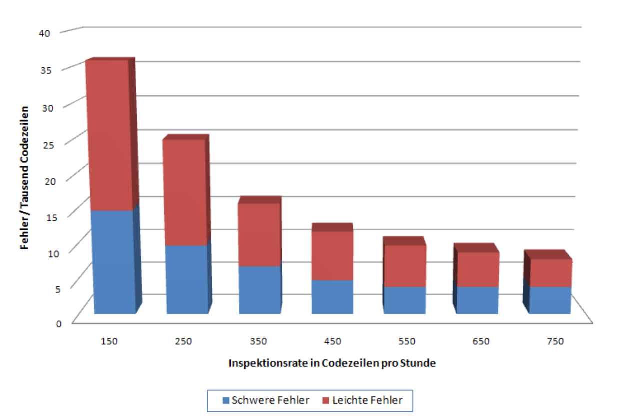


Abbildung 3: Anzahl der gefunden Fehler bei unterschiedlichen Inspektionsraten von Codezeilen. Vergleiche [14]

Wurde ein Dokument inspiziert und enthält noch Fehler, erfolgt die Fehlerbeseitigung (Rework) und ein abermaliges Review (Follow-Up).

Ein Problem von vielen Teilnehmern ist es, dass bei einer wiederholten Inspektion die überarbeitete Version von fehlerbehaftetem Code automatisch als fehlerfrei angesehen wird: „*Wird etwas verbessert, ist es fehlerfrei!*“. Als Faustregel gilt, dass jeder sechste Fehler nicht richtig korrigiert wurde oder im Verlauf der Verbesserung ein neuer Fehler eingeschleust wurde [4].

Aus Erfahrungen geht hervor, dass die Effektivität beim Prozess der Inspektion auf Codebasis durchschnittlich bei rund 60% liegt. Werden bei einem Dokument also dreißig Fehler gefunden, sind zirka zwanzig Fehler unentdeckt geblieben. Aus diesen dreißig Fehlern gehen fünf neue hervor, die die Produktivität einer Codeinspektion auf knapp 50% senkt.

Erst wenn die entdeckte Fehlerrate unter ein gewisses Niveau fällt (Beispiel: zwei leichte Fehler pro Seite) kann das Dokument weitergegeben werden und die nächste Phase kann beginnen.

Ein gewisses Restrisiko an Fehlern ist immer vorhanden. Ein einzelner Inspektionsdurchgang wird nie alle Fehler entdecken. In der Theorie könnte man Inspektionen so lange wiederholen, bis man keine Fehler mehr findet. Dies schließt jedoch nicht aus, dass keine Fehler mehr im Dokument vorhanden sind.

In der Praxis rechnet man mit einem gewissen Prozentsatz an Fehlern, deren Beseitigung den hohen Aufwand der Inspektionen im besten Fall unterschreitet. Die NASA setzt für kritische Systeme die wiederholte Durchsicht eines Dokuments von mehreren Inspektionsteams mit keinen gefundenen Fehlern voraus. Trotzdem kommt es, aufgrund von Fehlern in der Software, immer wieder zu Zwischenfällen.

2.4 Weitere Reviewtechniken

Walkthroughs sind Reviews informeller Art. Es müssen nicht alle Teilnehmer vorbereitet sein. Es gibt keine Checklisten und die Anberaumung einer Sitzung kann kurzfristig erfolgen. Sie werden unter anderem dazu verwendet, aktuelle Design- oder Programmfortschritte Mitarbeitern oder Managern zu präsentieren. Ebenso können Walkthroughs als Mitarbeitertraining benutzt werden. Obwohl auch hier Fehler gefunden werden können, ist dies nicht der primäre Fokus. Walkthrough-Sitzungen können durchaus zwanzig oder mehr Personen umfassen. Diskussionen sind hier, im Gegensatz zur Vorgehensweise bei Inspektionen, erwünscht.

Die Rolle des Präsentators kann gleichzeitig der Autor des Dokuments sein, der bestimmt welche Teile seiner Arbeit er zeigen möchte. Weitere Rollen sind nicht definiert.

Walkthroughs können in unregelmäßigen Abständen stattfinden. Durch ihren informellen Charakter können sie leichter organisiert beziehungsweise abgehalten werden. Inspektionen neigen hingegen dazu, seltener durchgeführt zu werden. Die Administration für ein Inspektionsmeeting ist aufwendig und muss auf die einzelnen Mitglieder abgestimmt werden.

Eine weitere Form des Reviews ist das Audit. Bei einem Audit stehen die Einhaltung der verwendeten Prozesse und deren Güte im Vordergrund. Nicht die Implementierung an sich. Organisationen, die solche standardisierten Prozesse verwenden, können sich dementsprechend zertifizieren lassen. Für den Einsatz von Qualitätsmanagement gibt es unter anderen die ISO 9001 Zertifizierung. Diese gibt allerdings keine Auskunft über die Qualität eines Produkts, sondern nur, dass bestimmte Abläufe während der Qualitätssicherung eingehalten werden.

Der Sinn hinter der Standardisierung von Prozessen sind einheitliche und kontrollierbare Abläufe: Was muss im Falle einer Krise getan werden? Wer muss informiert werden? Welche Dokumente müssen dabei erzeugt werden? Welche Workflows müssen angestoßen werden?

Es gibt zwei Arten von Audits: interne und externe. Interne Audits werden benutzt um Probleme im internen Prozessablauf zu finden und zu optimieren. Externe Audits werden in der Regel von einem unabhängigen Personenkreis (meist von akkreditierten Unternehmen) durchgeführt. Für eine ISO Zertifizierung müssen beide Arten des Audits in regelmäßigen Abständen durchgeführt werden.

2.5 Inspektion in Aktion

Zehn Jahre nach der Veröffentlichung seines ersten Artikels hat Fagan die nachfolgende Grafik (Abbildung 4) vorgestellt [4]. Sie beschreibt die Projektdauer einer durchschnittlichen Softwareentwicklung mit und ohne Inspektionen. Das Ergebnis zeigt, dass bei Projekten mit Inspektionen Verkürzungen der Projektlaufzeit und Einsparungen bei Mitarbeiterressourcen möglich sind. Den Mehraufwand für Inspektionen, die zirka 15% des Gesamtbudgets ausmachen, mit eingerechnet.

Inspektionen senken nicht nur die Kosten. Sie erhöhen auch die Qualität der Software. Im Vergleich zu ähnlichen Projekten konnten bei IBM und AETNA Versicherungen 93%, respektive 82%, aller Fehler durch professionelle Inspektionen im Life-Cycle erkannt und frühzeitig eliminiert werden. Zudem kamen noch Einsparungen von geschätzten 85% an Programmierstunden, da schwere Fehler nicht erst in

der Testphase, sondern durch Inspektionen früher entdeckt wurden.

Zu dem gleichen Schluss kommt Russell [14]. Bei Bell-Northern Research wird im Durchschnitt ein Fehler pro aufgewendeter Mannstunde durch Inspektionen gefunden. Wohingegen 34 Mannstunden gebraucht werden um einen Fehler, der vom Kunden gemeldet wurde, auszubessern. Pro gefunden Fehler ergibt sich theoretisch eine durchschnittliche Einsparung von fast 33 Mannstunden.

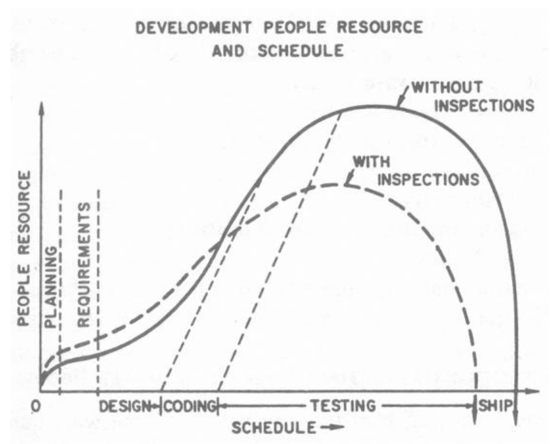


Abbildung 4: Reduktion von Ressourcen in Softwareprojekten mit und ohne Inspektionen

Des Weiteren stellt Russell zwei vergleichbare Projekte gegenüber: das erste wurde noch vor der Einführung, das zweite nach der Einführung von Inspektionen bei BNR umgesetzt. Beim ersten Projekt konnte durch den Systemtest eine Quote von rund 0,3 Fehlern pro Mannstunde entdeckt werden. Im Designtest lag die Quote lediglich bei der Hälfte. Beim zweiten Projekt konnten durch Inspektionen immerhin rund 0,6 Fehler entdeckt werden.

Russell kommt zu dem Schluss, dass unter optimalen Umständen Inspektionen bis zu zwanzigmal effektiver sind als das Testen. Das Testen vor Inspektionen ist also wesentlich ineffektiver als umgekehrt. Das Testen wird produktiver, wenn weniger Fehler vorhanden sind. Unterstützung bekommt Russell von Edward Weller von Bull Information Systems [15]. Weller bestätigt durch eigene Erfahrungen Russells Aussage indem er einige Nachteile von Testen vor Inspektionen aufzählt:

- Testen senkt die Motivation beim Inspektionsteam. Getestete Komponenten funktionieren und lassen den Eindruck entstehen, dass sie korrekt sind
- Codeinspektionen können Fehler aus einer vorangegangenen Phase aufdecken, die normales Modultesten nicht entdecken kann. Wird vor der Inspektion getestet, muss gegebenenfalls der gesamte Aufwand des Testens wiederholt werden
- Mitunter können die Ergebnisse aus Inspektionen vor dem Modultest so gut sein, dass diese Testphase übersprungen werden kann und stattdessen gleich ein Integrationstest durchgeführt werden kann. Dies würde Zeit und andere Ressourcen einsparen.

3. Moderne Tools

Fagan führt in seiner Publikation aus, dass die Vorbeugung von Fehlern weitaus effizienter sei [4]. So konnten Firmen, die nur einen Prozent des Gesamtbudgets in die Verhinderung von Fehler investierten, zum Beispiel in Schulungen, die Kosten von der Fehlerbeseitigung um das sechs bis achtfache reduzieren [2].

Moderne Werkzeuge können Entwicklern bei der Fehlervermeidung helfen. In den letzten Jahren hat sich vieles rund um den Softwaremarkt getan. Code kann während der Eingabe überprüft werden, Compiler optimieren Code und Entwicklungsumgebungen helfen dem Programmierer bei seiner Arbeit.

3.1 PSP

Der Persönliche Softwareprozess (PSP) wurde 1995 von Humphrey eingeführt [19]. Dabei handelt es sich um eine Ansammlung von Methoden und Techniken zur Verbesserung der individuellen Projektplanung und kontinuierlichen Verbesserung der eignen Arbeiten. Ziel ist es, die persönliche Effektivität zu erhöhen, genauere Aufwandsschätzungen für Projektteile geben zu können und seine Arbeit besser zu planen.

Dabei ist jeder aufgefordert seine eignen Aufzeichnungen in allen Bereichen seiner Arbeit zu führen. Aus diesen lassen sich Stärken und Schwächen ablesen, die man im individuellen Prozess fördert beziehungsweise zu vermeiden lernt.

Ein Beispiel hierfür ist die Selbstinspektion: der Softwareentwickler ist angehalten seine Arbeit ständig zu inspizieren und begangene Fehler festzuhalten. Aus diesen Aufzeichnungen können bestehende Makel abgelesen und gegebenenfalls durch Schulungen in Zukunft vermieden werden.

Ein Nachteil vom PSP ist der enorme administrative Aufwand jedes Einzelnen. Studien haben jedoch gezeigt, dass der Einsatz von PSP langfristig die Nachteile überwiegt.

3.2 Entwicklungsumgebungen

Entwicklungsumgebungen (oder IDE: Integrated Development Environment) sind grafisch unterstützte

Anwendungen, in denen der Benutzer seinen Code programmiert. IDEs gibt es inzwischen für viele Programmiersprachen. Die bekanntesten IDEs sind Eclipse für fast alle gängigen Sprachen und IntelliJ IDEA speziell für Java [9, 10].

IDEs erhöhen die Arbeitskraft eines Programmierers um ein vielfaches. Alle größeren IDEs unterstützen den Autor beim Schreiben, Modifizieren, der Fehlersuche und vieles mehr. Sie unterstützen Code Completion, Refactoring (automatisches umbenennen von Variablen, Methoden, Klassen und deren Verwendungen), automatische Generierung von Methoden, Code Beautifier (korrektes Einrücken von Blöcken, Verzweigungen und Methoden) und zeigen dem Benutzer ‚*live*' eventuelle Fehler.

In weiterer Folge möchte ich kurz JetBrains' *IntelliJ IDEA* vorstellen, da diese IDE eine große Palette von eingebauten Code Analyse Tools und Verifikationsmöglichkeiten bietet.

In IDEA findet man unter dem Menüpunkt ‚*Analyze*' den Punkt ‚*Inspect Code*'. Hier kann ein erfahrener Benutzer über sechshundert verschiedene Einstellungen treffen, anhand deren der Code analysiert werden soll. IDEA überprüft auf Wunsch einzelne Dateien oder das gesamte Projekt. Einige sinnvolle Beispiele sind unter anderem:

- ‚Dead Code' – unausgeführte Codeteile oder Methoden; nicht verwendete Variablen; Interfaces, die keine implementierten Klassen besitzen
- Redundante Importe, Deklarationen oder Casts
- Doppelte Zuweisungen ohne ein Auslesen
- Doppelte Deklaration von globalen und lokalen Variablen
- Variablen, die nur lokal verwendet werden, aber globaler definiert wurden
- Unnötige arithmetische Funktionen – Addition, Subtraktion oder Division mit Null
- Unperformanter Code: manuelles Kopieren von Arrays oder wenn Strings in Schleifen zusammengebaut werden
- Namenskonventionen – Großschreibung bei Konstanten; String Variablen, die ‚integer' heißen

Zusätzlich zur ‚*live*' Prüfung (Tool-Tips), stellt die Inspektion des Codes im Beispiel von Abbildung 5 fest, dass die Variable *i* von der Funktion *calc()* immer *10* beträgt, die Methoden *calc()* und *run()* auch *private* sein können und die Methode *run()* nie aufgerufen wird.

```
public static void main(String args[]) {
    calc(10);
}

public static int calc(int i) {
    if (i == 10) {
        return calc(10);
    } else {
        return 0;
    }
    System.out.println("I am not reachable!");
}   Unreachable statement

public int run() {
            Method 'run' recurses infinitely, and can only end by throwing an exception
    return run();
}
```

Abbildung 5: Geprüfter IDEA Code

Das abgebildete Programmbeispiel enthält aber einen schweren Fehler: die Methode *calc(10)* in *main()* wird sich immer wieder selbst aufrufen bis zum Programmabsturz. IDEA konnte diesen Zyklus nicht entdecken.

Ein weiterer Punkt in IDEA ist ‚Safe delete'. Möchte der Programmierer Methoden oder Klassen löschen, so prüft die IDE alle im Projekt enthaltenen Klassen auf Abhängigkeiten. Wird die zu löschende Methode oder Klasse anderswo verwendet, so wird eine dementsprechende Liste ausgegeben.

3.3 Agile Softwareentwicklung

Agile Softwareentwicklung hat sich in den letzten Jahren mehr und mehr durchgesetzt. Der Aufwand ist meist gering und unbürokratisch. Wenige, einfache Regeln erleichtern den Einstieg. Ein Beispiel hierfür ist Extreme Programming, auf das ich nicht im Ganzen eingehen möchte, sondern nur auf den Teil des Pair-Programmings.

Pair-Programming bedeutet, dass zwei Personen an einem Bildschirm arbeiten. Durch dieses Vier-Augen Prinzip wird der Code gleichzeitig geschrieben und geprüft. Fehler können sofort besprochen und ausgebessert werden.

Die vermeintlich höhere Softwarequalität aus dieser Methode wird in *A multiple case study on the impact of pair programming on product quality* diskutiert [11]. Darin werden vier Softwareprojekte gegenübergestellt, die jeweils alleine und mit Zweier-Teams entwickelt wurden. Dabei ist deutlich zu sehen, dass Paarprogrammierung bei der Produktivität, der Einhaltung der Codestandards, der kommentierten Bereiche (Anzahl und Qualität) und Softwarequalität stets vor der Einzelprogrammierung liegt. Zum Teil liegt die Anzahl der Fehler bei Soloprogrammierern beim sechsfachen Wert.

Im Gegensatz dazu werden einfache Programmteile alleine schneller und effektiver bewältigt. Komplexe Teile werden von Paarprogrammierern bis zu doppelt so schnell abgehandelt.

Die Studie kommt zu dem Schluss, dass gerade in der Anfangszeit eines Projekts die Paarprogrammierung von Vorteil ist. Zu zweit ist der Lernfaktor und die Motivation wesentlich höher und die Einarbeitungszeit wesentlich geringer als alleine.

Die von McDowell, Werner, Bullock und Fernald vorgestellte Untersuchung [12], die die Noten von rund sechshundert Studenten in Programmiereinstiegskursen untersuchte, zeigte ebenfalls die bessere Qualität und Programmierleistungen bei Paaren. Allerdings konnten die Einzelprogrammierer bei der Abschlussprüfung die knapp besseren Noten vorweisen. Andererseits war die Motivation bei Paaren, den Kurs bis zum Ende zu besuchen wesentlich höher: rund 92% der Paare und nur 75% der Einzelprogrammier gingen zur Endklausur.

Phongpaibul und Boehm stellten eine empirische Studie aus Thailand vor [16]. Sie untersuchte die Unterschiede zwischen dem Einsatz von Paar-Entwicklung und dem Einsatz von Inspektionen anhand von zwei Projekten. Da die Projekte eher kleineren Ausmaßes waren (600 Mannstunden beziehungsweise 1400 Mannstunden) sollte man die Ergebnisse nicht überbewerten oder sie auf größere Projekte umlegen. Sie geben aber einen kleinen Einblick in die Möglichkeiten von Paarprogrammierung.

Beim ersten Experiment wurde von mehreren Studententeams (sieben Teams haben Paarweise entwickelt; sechs individuell mit Inspektionen) in etwa die gleiche Qualität erzielt, bei rund einem Viertel weniger Entwicklungskosten.

Das zweite Experiment stellte zwei vergleichbare Softwareentwicklungen eines großen thailändischen Unternehmens gegenüber. Wieder arbeitete ein Team mit Inspektionen. Das zweite mit Paarprogrammierung. Die Entwicklungskosten beider Projekte waren zirka identisch. Jedoch war die Qualität des zweiten Projekts weitaus besser. Es wurden um 40% weniger Fehler gemacht als beim Inspektionsteam.

3.4 Reuse

Die Vision, Software nicht mehr von Grund auf neu zu entwickeln, sondern wie Bausteine aus bereits existierenden Komponenten zusammenzusetzen, existiert bereits seit beinahe vierzig Jahren. Damals geriet die Softwareindustrie in eine scheinbar ausweglose Situation: kosteneffektive Software zu produzieren war aufgrund von konstanten Expertenmangel kaum möglich. McIlroy stellte 1968 auf einer Konferenz seinen Artikel *Mass Produced Software Components* vor [20]. Darin beschreibt er wie man Basisbibliotheken, wie I/O Methoden, mathematische Funktionen und dergleichen, wiederverwenden könnte. Trotz einer breiten Akzeptanz konnte sich Softwarereuse bis heute als etablierter Standard nicht durchsetzen. Erst in den letzten Jahren entstand rund um die Wiederverwendung von Softwarekomponenten ein größer werdender Wirtschaftszweig.

Softwarewiederverwendung hat eine Reihe von Vor- und Nachteilen (vergleiche [17], S. 18-21).

Komponenten, die für die Wiederverwendung entwickelt werden, haben einen vermeintlich höheren Qualitätsstandard gegenüber solchen, die einmalig verwendet werden. Sie werden intensiver fehlerbereinigt und getestet. Durch den Einsatz wird die Produktivität erhöht, da weniger Code geschrieben werden muss und redundante Arbeiten und Entwicklungsphasen entfallen können. Der Zeitraum bis zur Marktreife wird reduziert und die Wartung von robusten Komponenten fällt geringer aus.

Dem gegenüber stehen Probleme wie das Auffinden einer geeigneten Komponente und allfällige Anpassungen an die eigenen Bedürfnisse. Sie sind in der Regel nur mit erhöhtem Aufwand möglich. Modifikationen an wiederverwendbaren Komponenten können ungewollte Auswirkungen auf andere Bereiche haben. Weiters müssen Ressourcen für die Suche, Integration und Test von Modulen bereitgestellt werden. Im Problemfall ist man, bei externen Anbietern von wiederverwendbarer Software, auf eine schnelle Fehlerbehebung angewiesen und kann nicht selbst tätig werden.

Eine Form der Wiederverwendung ist die komponentenbasierte Entwicklung wie sie beispielsweise in den Visual-Programmiersprachen von Microsoft Verwendung finden. Durch Drag & Drop kann der Entwickler vorgefertigte Komponenten wie Buttons, Menü-Einträge, Datei-Browser, Kalender, Auswahllisten und viele mehr in seine Applikation übernehmen. Durch die Common Language Infrastructure (CLI) – einem Standard von Microsoft – ist es sogar möglich Komponenten, die in verschiedenen Sprachen programmiert wurden miteinander zu kombinieren [18].

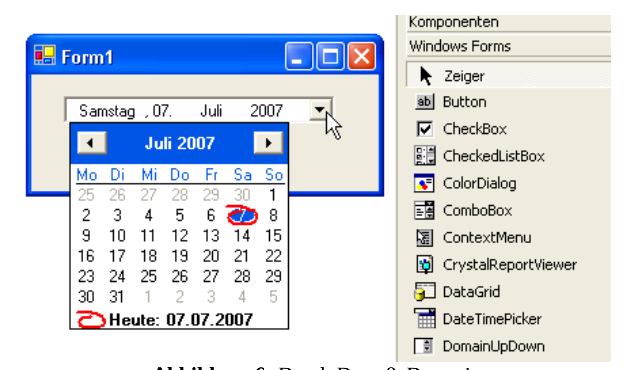


Abbildung 6: Durch Drag & Drop eingesetzter DateTimePicker ohne zusätzliche Programmierung

Dadurch wird ein wesentliches Problem, nämlich das Auffinden beziehungsweise die Verwendung der richtigen Komponente für die richtige Sprache gelöst.

4. Zusammenfassung

Die Entwicklung von Fagan war ein Meilenstein in der Softwareentwicklung. Durch die Einführung von Inspektionen konnte einerseits die Qualität der Software erhöht, andererseits auch die Kosten für Produktentwicklung und -wartung verringert werden.

Moderne Werkzeuge, wie sie unter Punkt 3 beschrieben werden erleichtern die Bearbeitung und Analyse von Code und senken die Fehlerrate beträchtlich. Sie vermindern zwar die Anzahl der Initialfehler. Können aber nicht alle entdecken, die ein geübtes Inspektionsteam zu finden vermag.

Daher sind diese Werkzeuge auf keinen Fall ein Ersatz für Code-Inspektionen. Sie sollten dennoch als vorbeugendes Instrument eingesetzt werden.

5. Referenzen

[1] M.E. Fagan: *Design and code inspections to reduce errors in program development*, IBM Systems J. Vol 15, No. 3, 1976, pp.182-211.

[2] Tom Gilb und Dorothy Graham: *Software Inspection*, Addison-Wesley, 1993

[3] David A. Wheeler, Bill Brykczynski und Reginald N. Meeson, Jr.: *Software Inspection – An Industry Best Practice*, IEEE Computer Society, 1996

[4] M.E. Fagan: *Advances in Software Inspections*, IEEE Transactions of Software Engineering, Juli 1986

[5] Boehm, Barry W.: *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, 1981

[6] National Aeronautics and Space Administration: *Software Formal Inspection Standard*, Approved 1993

[7] Michael Fagan Associates, http://www.mfagan.com, letzter Zugriff, 22. Mai 2007.

[8] Karl Wiegers: Seven Deadly Sins of Software Reviews, Process Impact, http://www.processimpact.com/articles/revu_sins.html, letzter Zugriff, 21. Mai 2007

[9] Eclipse Project, http://www.eclipse.org

[10] Jetbrains S.R.O, http://www.jetbrains.com

[11] Hanna Hulkko und Pekka Abrahamsson: *A multiple case study on the impact of pair programming on product quality*, ACM Press, 2005; International Conference on Software Engineering; Proceedings of the 27th international conference on Software engineering; Agile Sessions – Pages 495-504

[12] Charline McDowell, Linda Werner, Heather Bullock und Julian Fernald: *The Effects of Pair-Programming on Performance in an Introductory Programming Course*, ACM Press, 2002; Technical Symposium on Computer Science Education; Proceedings of the 33rd SIGCSE technical symposium on Computer science education; CS Session – Pages 38-42

[13] Bohm, Barry W.: *Developing Small Scale Application Software Projects: Some Experimental Results*, IFIP 8th World Computer Congress 1980

[14] Russell, Glen W.: *Experience with Inspection in Ultralarge-Scale Developments*, IEEE Software Vol. 8, No. 1, Jan. 1991, pp 25-31

[15] Weller, Edward F.: *Lessons from Three Years of Inspection Data*, IEEE Software Vol. 10, No. 5, Sept. 1993, pp 38-45

[16] Monvorath Phongpaibul und Barry W. Boehm: *An Empirical Comparison Between Pair Development and Software Inspection in Thailand*, ACM Press, Proceedings of the 2006 ACM/IEEE international symposium on empirical software engineering, 2006

[17] *C.R.U.I.S.E – Component Reuse In Software Engineering*, C.E.S.A.R e-Books, 2007, http://cruise.cesar.org.br, Letzter Zugriff: 14.07.2007

[18] Standard ECMA-335, *Common Language Infrastructure*, 4th Ed. Juni 2006, ECMA International 2006, http://www.ecma-international.org

[19] Humphrey, W.S.: *A Discipline for Software Engineering*, Addison-Wesley, 1995

Über Wissensrepräsentation zu Anforderungsmodellen

Martin Andritsch
0060102
m2andrit@edu.uni-klu.ac.at

Abstract

Ausgehend von einem Artikel aus dem Jahre 1982 von Sol J. Greenspan [1], in dem er die Wichtigkeit der ausreichenden Wissensansammlung bzw. in weiterer Folge deren Strukturierung und Repräsentation im Bereich des Requirements Engineerings betonte, möchte ich mit meiner Arbeit versuchen, die fortwährende Bedeutung des Wissensmanagements bis hin zu heutigen Softwareentwicklungen aufzuzeigen.
Es wird kurz generell auf den „Problembereich" Requirements Engineering eingegangen, um die Motivation der Anwendung einer Informationsmodellierung zu verdeutlichen. Dann wird auf die Aspekte des „world modeling" eingegangen gefolgt von einer kurzen Vorstellung einer von Greenspan entworfenen Sprache zur Wissensrepräsentation, dem RML. Weiters möchte ich heutige Ansätze bezüglich Wissensrepräsentation und -modellierung im Bereich Software Engineering erläutern. Im Speziellen geht der Artikel ein, auf die Wandlung vom objektorientierten Ansatz, aus dem auch RML entstand, hin zum zielorientierten. Auch der Einfluss der Künstlichen Intelligenz durch den Einsatz von Ontologien wird behandelt.

1. Einleitung

In der Einleitung in meine Arbeit möchte ich kurz generell Bezug auf den Problembereich des Requirements Engineering (RE) im Rahmen der Softwareentwicklung nehmen, um die Motivation von Anforderungsmodellen und Wissensrepräsentation aufzuzeigen. Im zweiten Abschnitt wird das Prinzip der Informationsmodellierung erläutert bzw. der Einfluss des Bereichs der Künstlichen Intelligenz deutlich gemacht.
In Abschnitt 3 wird eine von Sol J. Greenspan entwickelte Anforderungsmodellierungssprache als darstellendes Beispiel präsentiert. Abschnitt 4 wird den Übergang vom objektorientierten Ansatz aus den 80er Jahren hin zum zielorientierten Ansatz beschreiben. Abschnitt 5 beschäftigt sich mit Entwicklungen in der Künstlichen Intelligenz, speziell im Bereich der Ontologien. Im letzten Teil erfolgt eine Zusammenfassung über die behandelten Themen.

1.1. Problembereich Requirements Engineering

Vor 25 Jahren, als Sol J. Greenspan den Artikel [1], meinen Ausgangsartikel, verfasste, waren die Probleme, die im Zuge der Anforderungsanalyse bei der Entwicklung eines Softwaresystems entstehen können, bereits erkannt. Ausführliche Studien ergaben, dass über 70% aller Projekte schon damals ihre Ziele verfehlten oder überhaupt abgebrochen wurden aufgrund verschiedenster Fehler in der Findung und Definition von Anforderungen. Bis heute besteht das Problem der inkonsistenten, mangelhaften, unvollständigen oder mehrdeutigen Anforderungsdefinitionen. Je später im Produktentwicklungsprozess solche fehlerhaften Anforderungsdefinitionen entlarvt werden, desto höhere Kosten verursachen sie. Weiters können sie auch zu einer relevanten Verzögerung der Produktfertigstellung verursachen bzw. im schlimmsten Fall sogar zum Projektabbruch. Das Bewusstsein entstand, dass man dieser frühen Phase der Softwareentwicklung mehr Aufmerksamkeit schenken musste und dass es beim Definieren von Anforderungen um mehr geht, als um das Auffinden und Spezifizieren der funktionalen Anforderungen. Man erkannte, dass es von großer Bedeutung sei, sich ausreichend – wie es Sol J. Greenspan nannte – „real world knowledge", also Wissen über den Problem-/Anwendungsbereich anzueignen und zu verstehen. Dies umfasst unter anderem die Organisation und das System, in dem die fertige Software eingesetzt werden soll. Es wäre zum Beispiel bei der Entwicklung eines Informationssystems für ein Krankenhaus wichtig, sich mit einer Reihe von Inhalten, die ein Krankenhaus betreffen, vertraut zu machen. Wie z.B. medizinisches Wissen, Krankenhausabläufe, Therapiemöglichkeiten usw. [1].

In weiterer Folge muss dieses Wissen strukturiert, organisiert und in dieser Phase von Details abstrahiert werden, um auch allen Stakeholdern verständlich zu sein. Schließlich soll auch ein anderes großes Problem des Requirements Engineerings (RE) gelöst werden – das der mangelnden Kommunikation mit den Benutzern.

2. „Modeling the World“

Die Konstruktion einer abstrakten, interpretierbaren Beschreibung ist also eine fundamentale Aktivität im Requirements Engineering geworden [4] und lässt sich auch mit den Begriffen Informationsmodellierung oder konzeptuellen Modellen formulieren. Unter dem Begriff Informationsmodellierung versteht man die Konstruktion von computerbasierten, symbolischen Strukturen, die einen bestimmten Bereich der Welt – den Problembereich – abbilden [3]. Diese Strukturen können auch als Wissensbasis genutzt werden, die über eine lange Zeitperiode hinweg weiterentwickelt wird und als „Wissens-Repository“ immer wieder verwendet werden kann, z.B. um die allfällige Wartung des Produktes zu erleichtern [3]. Über die Jahre wurden hunderte solcher Notationen und Anforderungsmodellierungssprachen entwickelt, die diese Aufgabe erfüllen sollen, von denen allerdings eine Vielzahl nur in einem einzigen Projekt angewandt wurde [10].
Ich möchte an dieser Stelle auch den Unterschied zwischen einer Modellierungssprache, wie sie oben erwähnt wurde, und einer Spezifikationssprachen feststellen. Spezifikationssprachen haben eine hoch entwickelte, mathematische Formulierung als Grundlage, z. B. *Z*, und dienen als direkte Vorstufe der Implementierung, während Anforderungsmodellierungssprachen ein erstes abstraktes, formales Modell des Problembereichs liefern und sich somit weit vor der Spezifikation anzusiedeln [5].
Zusammenfassend lässt sich sagen, dass die Idee des „World Modeling“ die ist, Entitäten, Aktivitäten und andere Phänomene als Objekte in einem Modell zu erfassen.

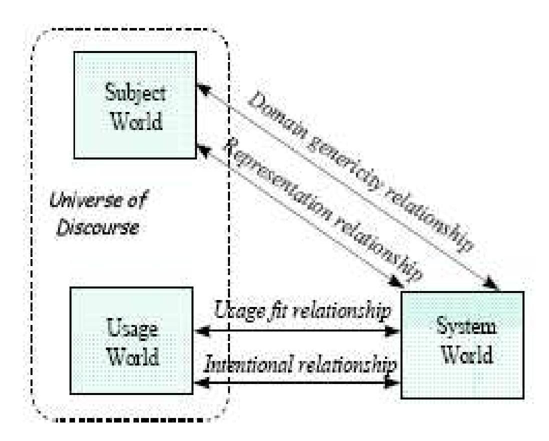


Abb. 1.: Aufteilung der “Welten“ [6].

Weiters kann nach [6] der Anwendungsbereich („Universe of Discourse) in zwei „Welten“ geteilt werden – der „Usage World“ und der „Subject World“.
Die „Usage World“ beschreibt die Aufgaben, Prozeduren und Interaktionen mitunter der Stakeholder, direkten und indirekten Benutzern.
Die „Subject World“ beinhaltet Wissen über den „real world“-Bereich, also existierende Objekte des Anwendungsbereichs, welche in einem konzeptuellen Schema dargestellt werden [6].
Schließlich gibt es eine dritte Welt, die „System World“, die die Systemspezifikation darstellt und den Anforderungen, die aus den beiden anderen Welten entstehen, gerecht werden muss [6].

2.1. Informal, semi-formal oder formal

Eine weitere Kernfrage des RE ist, welcher Grad der Formalisierung beim Einsatz von Anforderungs- bzw. Wissensmodellen angewandt werden soll. Tatsache ist, dass Modelle als Kommunikationsmedium zwischen Kunden und Entwicklern dienen und somit allen Stakeholdern verständlich sein müssen, was für eine informale Notation spricht. Andererseits haben formale Sprachen für die Entwickler den Vorteil, dass sie die strukturelle Grundlage für die darauf folgenden Phasen des Designs und der Implementierung bieten können. Zusätzlich können sie mittels so genannter Analyseprogramme leichter auf Korrektheit und Konsistenz geprüft werden. Was sind jetzt eigentlich formale Modellierungssprachen? Eine Notation ist formal, wenn sie eine formale Menge an Regel beinhaltet, die ihre Syntax und Semantik definiert. Genauer betrachtet beinhalten formale Notationen

- **eine Ontologie** (eine Menge von Vermutungen über die Natur der Anwendung die modelliert werden soll)

- **eine Terminologie** (Terme, um über die Anwendung sprechen zu können, z.B. Entitäten, Beziehungen usw.)
- **eine Sprache** (um Aussagen in der Notation formulieren zu können)
- **und Abstrakionsmechanismen** (um große Modelle strukturieren zu können).

Informale Notationen sind natürlichsprachiger Text oder einfache Graphen, beides mit minimaler Syntax, keiner Ontologie und keiner näheren Semantik. Trotzdem sind sie ein wichtiger Part der Anforderungsanalyse, weil sie aus der Interaktion mit dem Kunden hervorgehen und somit die eigentliche Wissensansammlung darstellen [11].

2.2. Geschichte der Informationsmodelle

Über die Jahre hinweg wurden die Informationsmodelle in 3 verschiedene Kategorien klassifiziert. Diese reflektieren eine historische Entwicklung weg von maschinenorientierten Repräsentationen hinzu zu den menschorientierten Modellen, die ausdrucksstärker sind und mit komplexen Anwendungsmodellierungsaufgaben besser zurecht kommen. Die drei Kategorien sind:

- **Physische Informationsmodelle:** diese Modelle benutzen konventionelle Datenstrukturen und andere Programmierkonstrukte um eine Anwendung zu modellieren. Der große Nachteil dieser Modelle war der Konflikt zwischen Recheneffizienz einerseits und der Qualität der Modellierung andererseits.
- **Logische Informationsmodelle:** verwendet abstrakte mathematische Formelstrukturen um Implementationsdetails vom Benutzer zu verbergen.
- **Konzeptuelle Informationsmodelle:** bietet ausdrucksstärkere Möglichkeiten, um Anwendungen zu modellieren und Informationen zu strukturieren. Als Beispiele seien semantische Ausdrücke wie `Entity` oder `Activity` angeführt. Darüber hinaus bieten solche Modelle Abstraktionsmechanismen, um Informationen auch organisieren zu können [3].

Bis heute werden konzeptuelle Modelle verwendet, um Anforderungen zu modellieren. Nur die Ansätze haben sich auch hier im Laufe der Zeit verändert. Mitte der 80er, mit dem Aufkommen objektorientierter Programmiersprachen, begann man auch objektorientierte Modellierungstechniken anzuwenden, die Eigenschaften aus eben der objektorientierten Programmierung, aus semantischen Datenmodellen und aus Anforderungssprachen mit sich brachten.

2.3. Die Rolle der Künstlichen Intelligenz

Wissensrepräsentation hat ihren Ursprung im Bereich der Künstlichen Intelligenz, wo ein zentrales Thema darin besteht, genügend Wissen anzusammeln, um u. a. menschliches Verhalten simulieren zu können. Daher dient dieser Bereich als Quelle von Ideen und Techniken, um Wissen zu repräsentieren und somit die Entwicklungen in anderen Bereichen der Informatik zu verbessern. Solche Techniken können neue Ansätze liefern, um z.B. Wissen anzusammeln, formal festzuhalten, organisieren und abzurufen [2].
Natürlich können diese Methoden nicht einfach übernommen werden, sondern müssen für jeden Bereich adaptiert werden.
Im Speziellen liegt der Einfluss der Künstlichen Intelligenz in den Bereichen:

- **neuer Konzepte (Ontologien)** – um das Wissen zu repräsentieren
- **neuer Notationen/Sprachen** – um die Software unter einem gewissen Grad von Abstraktion zu spezifizieren
- **besserer Semantik** – für bestehende Anforderungen und Designsprachen
- **Software Wissensbasis** – welche nutzvolles Wissen über ein spezielles Softwaresystem über den gesamten Lebenszyklus beinhaltet, verständlich organisiert ist und Abfragemechanismen bereitstellt [2].

3. Anforderungsmodellierung mit RML

In diesem Abschnitt werde ich die formale Anforderungsmodellierungssprache RML (Requirements Modeling Language) vorstellen. Sie wurde 1984 von Sol J. Greenspan auf der Basis eines früheren, von Greenspan entworfenen Frameworks entwickelt und war als formale Grundlage der SADT (Structured Analysis and Design Technique)-Diagramme gedacht. Somit schließt sie eine gleichzeitige Anwendung einer informalen Notation nicht aus.
RML greift dabei auf Ideen aus der Wissensrepräsentation und den Semantischen Datenmodellen zurück. Das Resultat ist eine objektorientierte, formale Anforderungssprache, in der Entitäten und Aktivität in einer Generalisationshierarchie organisiert sind und die einigen anderen objekt-orientierten Techniken voran gegangen war [3].
Darüber hinaus soll eine Sprache wie RML als explizites Modell des Anwendungsbereiches einschließlich

des organisatorischen Umfelds gesehen werden, in dem das benötigte Wissen strukturiert und organisiert wird, und das so den Entwicklern als Verständnisgrundlage dienen kann.

3.1. Abstraktionsmechanismen

Abstraktionsmechanismen sind beschreibende Einrichtungen, die es erlauben, bestimmte Informationen zu inkludieren, während wiederum andere, in diesem Level weniger wichtige Informationen (Design- und Implementierungsdetails), vernachlässigt werden können.
In RML kommen die Mechanismen Aggregation, Generalisation und Klassifikation zum Einsatz. Die Aggregation erlaubt es, Entitäten als Zusammensetzung von Komponenten zu sehen. Die Klassifikation erlaubt gemeinsame Charakteristiken von Entitäten innerhalb einer Klasse. „Individuelle" Unterschiede werden dabei ignoriert. Durch die Generalisation wird die Einführung von Subklassen ermöglicht, die die Eigenschaften der Oberklasse erben.

3.2. Die „Features" von RML

In RML werden die Objekte der echten Welt mittels der drei Einheiten dargestellt: den Entitäten, Aktivitäten und Behauptungen. Das Prinzip des RML besteht jetzt darin, die oben genannten Abstraktionstechniken uniform auf alle drei Einheiten anzuwenden [5]. Das bedeutet, dass es Klassen für Entitäten, Aktivitäten und Behauptungen gibt. Jede dieser Klassen kann aus anderen Klassen einer anderen Einheit zusammengesetzt sein. Organisiert sind die Klassen einer Einheit durch die Anwendung der Generalisation bzw. Spezialisierung [1].
Klassen- und Metaklasseneigenschaften werden durch so genannte definitorische Eigenschaften beschrieben, die spezifizieren, welche Art Information deren Instanzen zugeordnet werden kann. Dadurch stellen sich auch die Beziehungen zwischen den drei Typen (Entitäten, Aktivitäten und Behauptungen) heraus.

```
Activity Class AdmitPatient  with
   participants
      pt: Person
      wrd: Ward
      d: Doctor
   parts
      document: GetInfo&CheckBloodPressure(of<->pt)
      checkIn: AssignBed(toWhom<->pt,  onWard<->wrd)
      record: Admission(who<->newPatient, where<->wrd, when<->today)
   preconditions
      isThereRoom?: wrd.admitted# <  wrd.capacity
      patientAlready?:  not(pt is in Patient)
      canAdmit?: HasAuthority?(who<->d, where<->wrd)
   postconditions
      incrementCount:  wrd.admitted# <-   wrd.admitted#  + 1
      makePatient:   pt is in Patient
end Activity

EntityClass Patient  with
   necessary, unique, parts
      record: MedicalRecord
   associations
      location: NursingHome
      room: Room
      physician: Doctor
   producers
      register:AdmitPatient(pt<->this)
   modifiers
      assessment:AssessPatient(patient<->this)
   consumers
      release:DischargePatient(patient<->this)
   initially
      startClean? : record.paymentDue = 0
end Patient
```

Abb. 2.: Aktivität und Entität in RML [5].

Das Beispiel der Klassenbeschreibungen einer Aktivität und einer Entität in Abb. 2 soll die erwähnten Techniken verdeutlichen. Es ist ersichtlich, dass die Art des beschriebenen Objekts durch das erste Schlüsselwort definiert wird. So beschreibt einen `ActivityClass` eine Aktivität usw. Weiters bestehen die zwei Klassenbeschreibungen aus einer Reihe von gruppierten Eigenschaftskategorien, von denen jede einzelne einer Aktivität, Entität oder Behauptung entspricht. Die Eigenschaftskategorie `parts` umfasst beispielsweise in der Aktivitätsklasse wiederum drei Aktivitäten. Eine Aktivitätsklasse enthält zusätzlich noch Pre- und Postconditions, die, nach bzw. vor dem Auslösen der Aktivität, zutreffen müssen.
Das eigentlich Neue im Vergleich zu anderen ähnlichen Frameworks und Anforderungsmodellierungssprachen war die Hinzunahme der Behauptungen in das Modell (siehe Abb. 3). Auch auf sie werden die Abstraktionsmechanismen angewandt. Dadurch können informale Zusätze aus anderen Modellen formal in das Modell eingebracht werden. Verwendet werden sie als Pre- und Postconditions in Aktivitäten bzw. als Invarianten in Entitätsklassen.

```
AssertionClass IsTreatedWith with
   arguments
      p: Patient
      t: Treatment
   parts
      c1:Available(tr<->t,at<->p.location)
      c2:Recommended(...)
end IsTreatedWith
```

Abb. 3.: Behauptung in RML [5].

Zusammenfassend gesagt ist RML eine formale Anforderungsmodellierungssprache, die Techniken aus den Bereichen Künstliche Intelligenz (Knowledge Engineering) und Datenbanken (Datenmodellierung, konzeptuelle Modelle, Aggregation, Generalisation) nützt, um einerseits eine Wissensbasis zu erstellen und andererseits als objekt-orientiertes Informationsmodell im Zuge der Softwareentwicklung zu dienen.
RML unterstützt drei verschiedene Arten von Objekten (Aktivitäten, Entitäten und Behauptungen), die miteinander durch binäre semantische Beziehungen verbunden, in Klassen gruppiert und durch Generalisierung/Spezialisierung organisiert sind. Solche Modellierungssprachen eignen sich gut für die stufenweise Spezialisierung [5]. Was RML ein wenig einschränkt, ist die fixe Betrachtung der Welt im Sinne einer fix eingebauten Notation. Diese erlaubt es nicht, bei Bedarf neue Notationen einzuführen, die Sprache anderen Gegebenheiten anzupassen und sozusagen Maß zu schneidern [5].

4. Entwicklung im Requirements Engineering

Folgende Abschnitte behandeln das Umdenken vom objektorientierten Ansatz hin zum zielorientierten Ansatz im Bereich des RE.

4.1. Von objekt-orientierten zu ziel-orientierten Methoden

Wie schon erwähnt, hatte der Einsatz objektorientierter Methoden in der Programmierung Anfang der 80er Jahre begonnen. Dies hatte zwangsläufig auch großen Einfluss auf das RE und führte zum Einsatz objektorientierter Modellierungsmethoden. Noch in dieser Zeit war die grundlegende Überlegung, *was* das zukünftige System zu leisten hätte. Methoden im RE waren dadurch von Programmierkonzepten inspiriert. Durch die Zunahme an Dynamik im geschäftlichen und organisatorischen Umfeld, für das die Systeme heutzutage hauptsächlich Lösungen bieten sollten, ging man daran, die Beziehung zwischen dem System und dessen Umgebung zu überdenken. Modellierungstechniken sollten nun imstande sein, auch das *Warum* in Betracht zu ziehen, um so die semantische Kluft zwischen dem System und dessen Umgebung so weit wie möglich zu schließen [12].

4.2. Vorteile durch Zielorientiertheit

Es gibt mehrere Gründe, warum eine Berücksichtigung von Zielen im Prozess des RE hilfreich sein kann. Zum einen wird das Streben nach Vollständigkeit der Anforderungsspezifikation in der Hinsicht unterstützt, dass eine solche Spezifikation nur dann vollständig sein kann, wenn alle Ziele durch die spezifizierten Eigenschaften des Systems erfüllt werden können. Auch irrelevante Anforderungen, nämlich solche, die nichts zur Erfüllung zumindest eines Zieles beitragen, können so erkannt werden. Weiters deckt eine Verfeinerung der Ziele mögliche Konflikte, die durch verschieden Betrachtungen entstehen können, auf.
Nicht unwesentlich erleichtert die Definition von Zielen die Kommunikation mit dem Auftraggeber. So können beispielsweise durch die Zielverfeinerung, die in einer Baumstruktur dargestellt ist, strategische Ziele mit technischen Anforderungen nachvollziehbar in Beziehung gebracht und so dem Kunden leichter verständlich gemacht werden. So können im Zuge dieser Zielverfeinerung mögliche Alternativen ausgeforscht werden, die Analytiker in der Folge gemeinsam mit dem Kunden ausarbeiten.

4.3. Vergleich mit objektorientiertem Ansatz

Im Vergleich zum objektorientierten Ansatz, erweitert der zielorientierte Ansatz ursprüngliche Ontologien um Notationen wie *Ziele, Abhängigkeiten, Rolle, Akteur und Position*. Aufgrund solcher zusätzlichen Notationen lässt sich die Frage nach der Intention einer Anforderung modellieren [5]. Darüber hinaus wird die traditionelle Anforderungsanalyse durch die Evaluierung und Berücksichtigung von Alternativen gestärkt [9].

4.4. Arten von Zielen

Man unterscheidet verschiedene Arten von Zielen:

- zu erreichende Ziele (achievement goals),
- zu bewahrende Eigenschaften (maintenance goals) und
- weiche Ziele (soft goals), die in der Regel nicht-funktionale Anforderungen betreffen.

Die Erfüllung der zu erreichenden Ziele bzw. der zu bewahrenden Eigenschaften kann durch formales

Schließen verifiziert werden, wohingegen die Erfüllung der „soft goals“ nicht eindeutig festgestellt werden kann (nur durch qualitatives Schließen) [14].
Zu erreichende Ziele werden mittels einer UND/ODER-Verfeinerung zueinander in Beziehung gesetzt was Abbildung 4. illustriert:

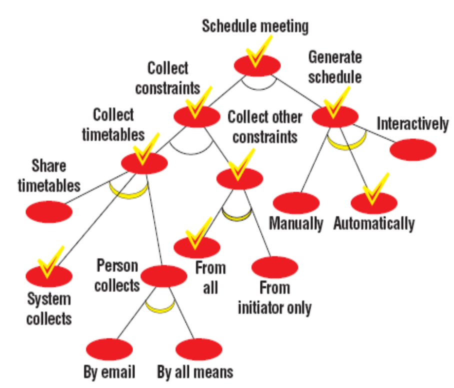


Abb. 4.: Zielverfeinerung mittels UND/ODER [9].

Untergeordnete Ziele, die durch ein „UND“ miteinander in Beziehung stehen (hier gekennzeichnet durch einen einfachen Bogen), müssen allesamt erfüllt werden, um das übergeordnete Ziel zu erfüllen. Übergeordnete Ziele, deren darunter liegende Ziele durch eine „ODER“ zueinander in Beziehung stehen (doppelter Bogen), verlangen nur nach der Erfüllung eines dieser „sub goals“.
„Softgoals“ hingegen, die nicht-funktionale Anforderungen darstellen, können nicht eindeutig erfüllt oder nicht erfüllt werden. Beispielsweise kann die Anforderung *„Das System muss in einem hohen Maße benutzerfreundlich sein“* keinen eindeutigen Kriterien zur Erfüllung oder Nichterfüllung unterzogen werden. Die Erfüllung solcher „Softgoals“ hängt davon ab, ob es mehr Indizien für die Erfüllung gibt, als dagegen (siehe Abb. 5.)[9].

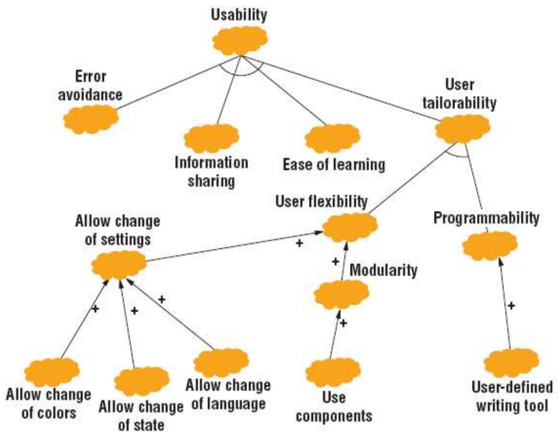


Abb. 5.: Zielverfeinerung von „Softgoals“ [9].

Schließlich werden die erarbeiteten und evaluierten zu erreichenden Ziele mit den weichen Zielen zusammengesetzt und so in eine Wechselbeziehung gebracht. Dadurch erkennt man, wie funktionale Anforderungen nichtfunktionale beeinflussen und umgekehrt. Abbildung 5. veranschaulicht dieses.

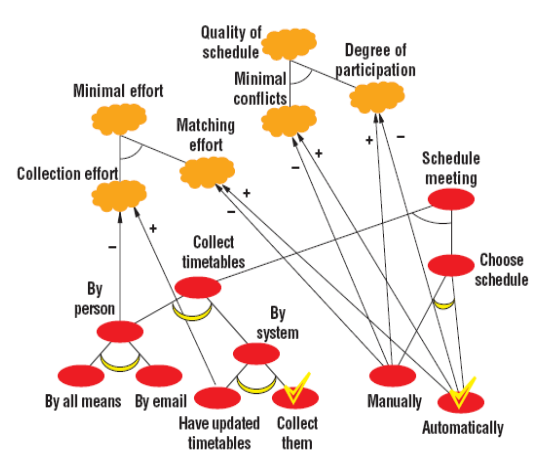


Abb. 6. Korrelation aller Ziele [9].

Für die Modellierung von Zielen wurden ebenfalls Frameworks in zahlreichen wissenschaftlichen Arbeiten vorgestellt. Im Zuge meiner Recherchen sind das „*i**“-Framework und die „KAOS“-Methode öfters aufgetreten.

5. Entwicklungen in der Künstlichen Intelligenz

Wie schon erwähnt hat die Wissensrepräsentation ihren Ursprung in der Künstlichen Intelligenz. Im speziellen liegt die Bedeutung im Einsatz einer Ontologie. In den folgenden Abschnitten möchte ich erklären, was Ontologie bedeutet und welche Entwicklungen es in diesem Bereich gegeben hat.

5.1. Was ist eine Ontologie?

Der Begriff „Ontologie“ wurde bisher bereits öfters erwähnt, aber was wird unter einer Ontologie verstanden?
An sich kommt der Begriff aus der Philosophie und bedeutet soviel wie die Lehre des Seienden und beschreibt somit die Beschaffenheit der Welt.
In der Informatik wurde der Begriff eingeführt, um einen Wissensbereich (knowledge domain) mit Hilfe einer standardisierenden Terminologie sowie von Beziehungen und Ableitungsregeln zwischen den dort definierten Begriffen zu beschreiben. In dieser Hinsicht gewinnen Ontologien in allen, mit Wissen befassten,

Bereichen der Informatik immer mehr an Bedeutung − ebenso in der Softwareentwicklung (SE), wo sie als Hilfsmittel und Wissensfundus dienen [15]. In der SE leitete die Einführung der Objektorientiertheit den Gebrauch von Ontologien ein. Man nannte diese Disziplin der Wissensbeschaffung und -analyse eines gewissen Bereiches der Welt „domain modeling“.

5.2. „Domain Model“

Wie im ersten Abschnitt bereits erläutert wurde, ist es bei der Entwicklung von Softwaresystemen äußerst wichtig, dass die Analytiker und Designer den Anwendungs- und Problembereich verstehen, vor allem jenen von großen und komplexen Systemen. Das so genannte „domain knowledge“ ist meist:

- informal statt formal,
- implizit statt explizit und
- nur unvollständig oder in problemspezifischer Sprache modelliert [17].

Ein „domain model“ kann nun als ein explizites, deklaratives, in formaler Sprache gehaltenes Modell über einen solchen Anwendungsbereich verstanden werden.

5.3. „Ontology Engineering“

Mit der steigenden Bedeutung von Ontologien betrieb man auch mehr Forschung über diese Materie und so entstand das „Ontology Engineering“ (OE). OE umfasst die Produktion von Ontologien, welche abstrakte Modelle eines Bereichs sind. Tatsächlich ist der Nutzen durch den Einsatz einer Ontologie nur so groß, wie dessen Fähigkeit, „Material“ (Gegenstände des Problembereichs) zu organisieren, worauf vorher noch nicht gestoßen wurde [16]. In anderen Worten: es muss möglich sein, den Inhalt einer Ontologie zu erweitern und zwar so, dass der Inhalt kosteneffektiv gefunden und verwendet werden kann.
Aus heutiger Sicht gibt es noch eine Reihe von ungelösten Problemen im Zusammenhang mit Ontologien [15]:

- Wie lassen sich gut verwertbare Metadaten für sehr große Ressourcenbestände erzeugen und konsistent weiterentwickeln?
- Können Ressourcen eindeutig klassifiziert werden?
- Wie lassen sich Metadaten in Ontologien einordnen?
- Macht es Sinn nach einer gemeinsamen Universalontologie zu suchen?

Forschungen untersuchen unter anderem Möglichkeiten, eine Notation für eine gemeinsame Ontologie zu entwickeln, die disziplinübergreifend in der Informatik Anwendung finden kann [15].
Aktuelle Bedeutung erlangten Ontologien im Zusammenhang mit der Entstehung des „Semantic Web“.

6. Zusammenfassung

In meinem Ausgangsartikel nahm sich Sol J. Greenspan der Probleme des RE an. Diese Probleme bestanden hauptsächlich in schlecht formulierten, inkonsistenten Anforderungen, welche zu unproportional kostenintensiven Korrekturen oder im worst case zu einem Projektabbruch führen können. In Abschnitt 3. verdeutlichte ich Greenspans Versuch, solche Fehler in der Anforderungsanalysephase mittels einer expliziten Wissensdarstellung zu verhindern. Er bediente sich dabei der Methoden aus den Bereichen der Datenbanken (Abstraktionsmechanismen) und der Künstlichen Intelligenz („Knowledge Representation“, Ontologien). In meinem Artikel wollte ich die nachhaltige Relevanz solcher expliziten Wissensdarstellungen im RE verdeutlichen.
Dazu erklärte ich zu Beginn Begriffe rund um die Informationsmodellierung und hob die Rolle der Künstlichen Intelligenz heraus. In Abschnitt 4. versuchte ich aufzuzeigen, dass die Berücksichtigung von Zielen in der Anforderungsanalysephase eine Anforderungsspezifikation weiter komplettieren kann. Weiters wurden Entwicklungen im Bereich der Künstlichen Intelligenz, speziell im Gebieten der Ontologien, dargestellt.

6.1. Warum formale Wissensrepräsentation heute noch Bedeutung hat

Heutzutage hat die standardisierte, semi-formale Modellierungssprache UML einen festen Platz in der Anforderungsmodellierung. Mit ihren zahlreichen Modellen lässt sich beinahe jede Problemstellung im Prozess der Softwareentwicklung abbilden. Formale Methoden zur Wissensdarstellung sind hingegen nicht leicht anzuwenden und auch kaum verbreitet. Mit steigender Komplexität und Größe des zu entwickelnden Systems bzw. in Bereichen, die höchste Präzision verlangen (z.B. Krankenhausbetriebe), können formale Sprachen zur Wissensrepräsentation und der Einsatz von Ontologien sehr unterstützend und unter Umständen zwingend notwendig sein. Sie liefern eine exakte unmissverständliche Wissensbasis, auf der weitere Modellierungen erfolgen können. Optimal wäre der Einsatz von beiden Methoden. So könnte eine semi-formale Modellierung der Kommunikation mit den Stakeholdern

dienen und auch für die Analytiker und Entwickler förderlich sein, während eine formale, den Modellen zugrunde liegende Wissensbasis eine exakt festgelegte Grundlage wäre, die bis zur Produktfertigstellung sich ändernden Anforderungen angepasst werden kann. Darüber hinaus würde eine solche Wissensbasis auch entscheidend zum Grad der Wartbarkeit des Systems beitragen.

7. Referenzen

[1] Sol J. Greenspan, John Mylopoulus, Alex Borgida, "Capturing More World Knowledge in the Requirements Specification", Proceedings of the 6th international conference on Software engineering, IEEE Computer Society Press , 1982

[2] John Mylopoulus, Alex Borgida and Eric Yu, "Representing Software Engineering Knowledge", Automated Software Engineering, Springer Netherlands, pp. 291-317, 1997

[3] John Mylopoulus, "Information Modeling in the Time of the Revolution", Information Systems 23, Elsevier Science Ltd, pp. 127-155, 1998

[4] Bashar Nuseibeh, Steve Easterbrook, "Requirements Engineering: A Roadmap", Proceedings of the Conference on the Future of Software Engineering, ACM Press, pp. 35-46, 2000

[5] Sol J. Greenspan, John Mylopoulus, Alex Borgida, "On Formal Requirements Modeling Languages: RML Revisted", Proceedings of the 16th international conference on Software engineering, IEEE Computer Society Press, pp. 135-147, 1994

[6] Colette Rolland, Naveen Prakash, "From Conceptual Modeling to Requirements Engineering", Annals of Software Engineering, CREWS Report Series, 1999

[7] Axel van Lamswerde, "Requirements Engineering in the Year 00: A Research Perspective", ICSE 2000, ACM, pp. 5-19, 2000

[8] Eric S. K. Yu, "Towards Modeling and Reasoning, Support for Early-Phase Requirements Engineering", Proceedings of the Third IEEE International Symposium, IEEE, pp. 226-235, 1997

[9] John Mylopoulus, Lawrence Chung, Stephen Liao Hauiquing Wang, Eric Yu, "Exploring Alternatives during Requirements Engineering", IEEE Software, IEEE, pp. 92-96, 2001

[10] John Mylopoulus, Lawrence Chung, Eric Yu, "From Object-Oriented to Goal-Oriented Requirements Analysis", Communications of the ACM, ACM Press, pp. 31-37", 1999

[11] J. Paul Gibson, "Formal Requirements Models: Simulation, Validation, Verification", Technical Report NUIM-CS-2001-TR-02, 2001

[12] Eric Yu, "Why Goal-Oriented Requirements Engineering", Proc. 3rd Int. Workshop on Requirements Engineering: Foundations of Software Quality REFSQ'97, pp. 171-183, 1997

[13] Alex van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour"

[14] Martin Wirsing, „Anforderungsmodellierung: Zielorientierte Methoden", Methoden des Software Engineering, Koch, Störrle, Wirsing, LMU München, 2004

[15] Wolfgang Hesse, „Ontologie(n)", *Informatik-Spektrum*, Springer Berlin/Heidelberg, pp. 477-480, 2002

[16] Dean Allemang, „Ontologies, Reuse and Domain Analysis", http://www.domainmodeling.com/Papers/domainmodeling_ontologies.pdf, (zugegriffen am 10. Juli 2007)

[17] Neil Iscoe, Gerald B. Williams, Guillermo Arango, "Domain Modeling for Software Engineering", Proceedings of the 13th international conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, pp. 340-343, 1991

Domain Engineering and Reuse

Ronny Haindl
9560276
rhaindl@edu.uni-klu.ac.at

Abstract

„Domain Analysis“ als Teil des „Domain Engineering“ ist seit Einführung der Ideen von Neighbor um 1980 ein aussichtsreiche Methode für die Wiederverwendung von Software Artefakten. Ziel ist die Steigerung der Produktivität und Qualität bei der Entwicklung von Software-Systemen. Dieser Artikel beschäftigt sich als Basis mit einem Artikel aus dem Jahre 1989 über einen systematischen Ansatz für Domain Analysis und Wiederverwendung von Software und untersucht die heutige Relevanz der damaligen Ideen und Konzepte. Dabei werden zwei Schwerpunkte gesetzt. Die Weiterentwicklung von Domain Engineering mit dem Fokus auf konkrete Produkte – „Product Line Engineering“ und zum Zweiten domänenspezifische Modellierung als Weiterentwicklung von komponentenorientierter Wiederverwendung.

***Keywords**: domain analysis, domain engineering, reuse, product line engineering, domain specific modeling*

1. Einführung

Die Entwicklung von Software-Systemen wird durch stetig steigende Anforderungen in unterschiedlichen Branchen komplexer und kostenintensiver. Viele Probleme erscheinen bei verschiedenen Systemen in ähnlicher Form. Mit Hilfe von Wiederverwendung von Software-Artefakten versucht man die Probleme schneller und einfacher zu lösen, um effizienter Systeme zu entwickeln.

Schon 1968 hatte McIlroy [2] die Idee, Systeme mittels industriell vorgefertigten Software-Komponenten zu implementieren. Das Wachsen der Komponentenbibliothek führte jedoch zu Schwierigkeiten, passende Komponenten zu finden. Ein zusätzliches Problem war die hohe Anzahl an ähnlichen Komponenten, die für spezifische Anforderungen angepasst und wieder der Bibliothek zugänglich gemacht wurden [3].

Arango, der Verfasser des Basisartikels [1], glaubt wie auch Neighbor [4] dass der Schlüssel für eine erfolgreiche Wiederverwendung in der Wiederverwendung von Analyse und Design liegt, nicht alleine in Code. Um effektive Wiederverwendung von Analyse und Design zu ermöglichen, muss dies in eingeschränkten Klassen von Systemen – in Domänen – passieren.

Gliederung

In Kapitel 2 wird der Basisartikel genauer betrachtet und die Ideen von Domain Analysis und Engineering von 1989 präsentiert. Kapitel 3 beleuchtet die Weiterentwicklung und Verfeinerung des Domain Engineering in Verbindung mit dem konventionellen Software Engineering. Einen Einblick in Domain Engineering mit dem Fokus auf Produktfamilien eines Unternehmens zeigt Kapitel 4. Domänenspezifische Modellierung in Kapitel 5 soll Möglichkeiten der Wiederverwendung auf höherer Abstraktionsebene als komponentenorientierte Entwicklung zeigen.

2. 1989 – Domain Analysis, Engineering, Wiederverwendung

Es besteht eine große Lücke zwischen dem informalen, impliziten Wissen einer Domäne und der notwendigen expliziten, formalen Form von wieder verwendbaren Informationen. Es gibt aber keinen systematischen Ansatz, diese wieder verwendbaren Informationen zu erstellen. Der Domain Engineering Prozess soll mit wohl definierten Aktivitäten und Zwischenprodukten diese Informationen erzeugen. Arango verwendet den Begriff „Wiederverwendungs-infrastruktur (reuse infrastructure), in der die notwendigen Informationen für den Wiederverwender zu Verfügung gestellt werden. Der DE Prozess beinhaltet 3 grundlegende Interessen [1]:

1) Domain Analysis: die Identifizierung, Erfassung und Evolution von wieder verwendbarer Information in einer Domäne. Diese Informationen werden in einem „Modell der Domäne“ zusammengefasst. Das Modell dient als Input für die nächste Aktivität.

2) Infrastruktur Spezifikation: die Selektion und Organisation der Informationen passend für die Umgebung der Wiederverwender. Als Ergebnis entsteht eine Architektur. Z.B. eine Programm-Bibliothek oder ein Datenbank-Schema.

3) Infrastruktur Implementation: die Architektur wird in der Zieltechnologie der Wiederverwender implementiert. Z.B. Implementierung von Programmen in Ada oder Lisp, indiziert mit einem Klassifizierungsschema.

Arango betrachtet hauptsächlich die Domänenanalyse und Infrastruktur Spezifikation, gibt aber dafür keine konkrete Methode vor, wie Domänenanalyse durchgeführt werden sollte. Die Ansicht ist auf einer Metaebene, in der unterschiedliche Aufgabenstellungen für Wiederverwendung möglich und daher auch unterschiedliche Methoden notwendig sind. Vielmehr gibt Arango Prinzipien eingebettet in einem Framework vor, die bei jeder Analysetätigkeit vorkommen [1].

Neighbor definierte eine allgemein akzeptierte Formulierung von Domain Analysis [4]:

“domain analysis is an attempt to identify the objects, operations and relationships between what domain experts perceive to be important about the domain.”

Arango glaubt, dass ein praktischer Ansatz von Domain Analysis im Gegensatz zum „puren künstlerischen“ Ansatz notwendig ist, da es keine präzise Definition von „wichtig“ und „Domäne“ gibt. Es gibt auch keine klaren Ziele und keine konkrete Formulierung, wann Domänenanalyse „fertig gestellt“ ist. Für den praktischen Ansatz formuliert Arango folgende Fragestellung [1]:

“How is a model of a domain incrementally constructed/evolved to achieve a specified level of perfomance with a given target reuser?”

Wiederverwender als Lernsystem

Der Fokus ist auf den Wiederverwender gerichtet, der in einem „Lernsystem“ eingebettet ist. Die Aufgabenstellung von Domain Analysis in diesem Lernsystem ist wie folgt [1]:

Finde eine Methode, die

1) Informationen in einer Domäne identifiziert, die in passender Form für den Wiederverwender einen bestimmten Grad an Effizienz für die Systemerstellung ermöglicht.
2) diese relevanten Informationen erfasst
3) diese Information verbessert, um die Effizienz zu erhöhen.

Das Lernsystem besteht aus der grundlegenden Aufgabenstellung des Wiederverwenders ein System zu erstellen. Er hat eine Spezifikation, die mit einer bestimmten Technologie implementiert werden soll. Dabei bedient er sich der Wiederverwendungs-infrastruktur, in der geeignete Informationen des Modells der Domäne in Form der passenden Technologie vorhanden sind. Die gesammelte Erfahrung eines Wiederverwenders während der Entwicklung fließt zurück in die Infrastruktur, um die Effizienz zu erhöhen. Neben dem Ziel der Effizienz sind auch andere Zieldefinitionen möglich. Z.B. Implementierungen zu erzeugen, die Speicher und CPU effizient nutzen oder Implementierungen, die sicher sind [1].

Mit mathematischen Konstrukten und Variablenbelegungen des Lernsystems können mit diesem Framework unter bestimmten Vorraussetzungen unterschiedliche Domain Analysis Methoden miteinander verglichen werden. Zusätzlich hat Arango auf Basis des Frameworks eine generische Strategie für die Entwicklung von Domain Analysis Methoden definiert [1].

In den folgenden Kapiteln werden im Überblick die Weiterentwicklungen von Wiederverwendung in Domänen betrachtet.

3. Evolution von Domain Engineering

Mit der intensiven Auseinandersetzung der Systementwicklung mit dem Fokus auf Domänen Anfang 1990 stieg das Verständnis der Beziehung zwischen Domain Engineering und System Engineering. Die drei von Arango beschriebenen grundlegenden Aktivitäten bleiben erhalten und werden in Beziehung zu System (Application) Engineering gesetzt. **Fehler! Verweisquelle konnte nicht gefunden werden.** zeigt die Verbindung zwischen Domain Engineering und Application Engineering. Dieses „2 Life Cycle“ Konzept zeigt den Unterschied zu der konventionellen System-entwicklung. Dort wird ein einziges System, bei Domain Engineering werden mehrere Systeme innerhalb einer Klasse betrachtet [5], [6].

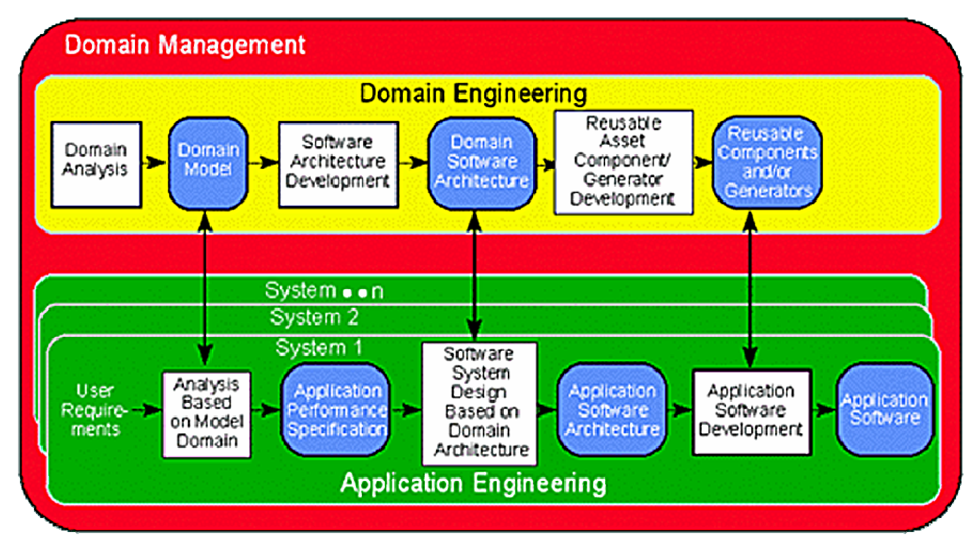


Abb. 1: Software Entwicklung basierend auf Domain Engineering [6]

Eine weitere Verfeinerung ist die Abgrenzung von Domänenarten und Beziehungen zwischen Domänen. Vertikale Domänen betreffen bestimmte Branchen z.B. Flugkontrollsysteme, Telekommunikation, etc. Horizontale Domänen betreffen Teilsysteme, die sich über mehrere vertikale Domänen erstrecken. Beispiele sind Datenbanksysteme oder Komponenten für das Finanzwesen [5].

Bei der Domänenanalyse ist die Ausarbeitung der Gemeinsamkeiten und Unterschiede zwischen den Systemen eine notwendige Vorgangsweise und findet größere Bedeutung bei den Produktlinien (Kapitel 4). Voraussetzung ist eine gewisse Stabilität der Domäne, sodass das Modell über längere Zeit gültig und korrekt bleibt [7], [8].

Für die Korrektheit des Models kommen Verifikations-, Validierungs- und Testmethoden zum Einsatz. Der Vorgang der Validierung versucht, das für die Interessensvertreter richtige Domänenmodell zu bestimmen, bei dem alle notwendigen Entitäten, deren Beziehungen und Zusammenhänge vorkommen. Die Verifikation sorgt für ein in sich korrektes und vollständiges Domänenmodell [9].

Beim Design der Architektur hat sich der Einsatz von Patterns bewährt, die die funktionalen und nicht-funktionalen Anforderungen abbilden. Besonderes Augenmerk muss dabei auf Adaptierbarkeit, Erweiterbarkeit, und Konfigurierbarkeit gelegt werden, sodass bei einer Entwicklung einer konkreten Applikation die wieder verwendbaren Artefakte bestmöglich eingesetzt werden können [5], [10].

Als Ergebnis der Implementierung der Architektur können Komponenten, Programmgeneratoren, Frameworks oder domänenspezifische Sprachen entstehen, die bei der Applikationsentwicklung als Werkzeuge genutzt werden.

Bei der Applikationsentwicklung werden während der Analysephase die Kundenanforderungen aus dem Domänenmodell herangezogen. Neue Anforderungen, die nicht im Domänenmodell abgebildet sind, werden gesondert betrachtet und evaluiert und fließen auch wieder in das Domänenmodell ein. Die System Architektur basiert auf der Architektur des Domain Engineering Prozesses und wird angepasst. Mit Hilfe der erzeugten Artefakte bei der Domain Implementierung wird das System erstellt [5].

Bekannte Domain Analyse Methoden sind:

FODA - Feature-Oriented Domain Analysis, entwickelt von Software Engineering Institute (SEI) (http://www.sei.cmu.edu/)

ODM - Organization Domain Modeling, entwickelt von Mark Simos of Synquiry Ltd.

DARE - Domain Analysis and Reuse Environment

Einer der ersten Methoden, die auch Arango beeinflussten, war „Draco“, entwickelt von J. Neighbors [11].

4. Software Produktlinien (SPL) – Product Line Engineering

Viele Unternehmen entwickeln Systeme innerhalb einer Domäne und bringen unterschiedliche Produktvarianten auf den Markt [12], [13]. Man spricht auch von „Produktfamilien“, bei denen die Sichtweise auf die Gemeinsamkeiten der Produkte gelegt ist, während der Begriff „Produktlinien“ aus der Sicht der Marketing Strategie kommt und die Unterschiede der Produkte als „Features“ betrachtet werden [5].

Interessant in diesem Zusammenhang ist, dass schon Parnas um 1976 von „Programmfamilien“ gesprochen hat [14]. Doch erst Mitte der 90er wurden konkrete Konzepte entwickelt und eingesetzt.

Der Einsatz von Domain Engineering, um diese Produktvarianten zu realisieren, hat Auswirkungen auf die Organisation und Geschäftsprozesse des Unternehmens, nicht nur auf die technischen Aspekte [12], [15], [16].

Die Motivation für den Einsatz von Produktlinien liegt zum Ersten in der Reduktion von Entwicklungskosten. Zu Beginn der Umsetzung gibt es zwar höhere Anfangskosten, aber je mehr Produktvarianten entwickelt werden, desto geringer werden die Gesamtkosten im Vergleich zu herkömmlichen Entwicklungsstrategien, bei denen fertige Systeme an neue Kunden angepasst werden [7]. Abb. 2: Wirtschaftlichkeit von SPL verdeutlicht dies.

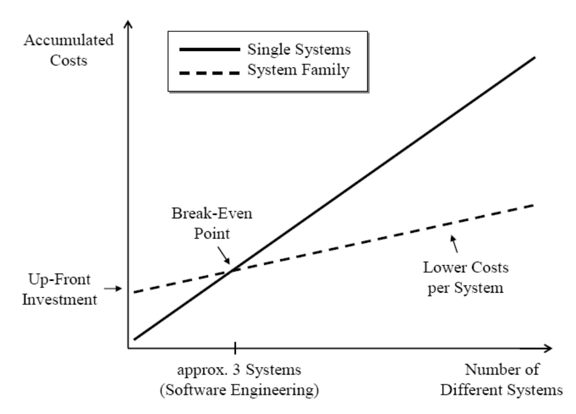


Abb. 2: Wirtschaftlichkeit von SPL [7]

Ein weiterer Grund ist die Reduktion der Zeit für eine Markteinführung einer Produktvariante. Durch die gemeinsame Basis der Produktlinie können schneller neue Produktvarianten erstellt werden [7].

Weiter Motivationsgründe sind die verbesserte Qualität, die Reduktion von Wartungszeiten, einfachere Weiterentwicklungen, bessere Handhabung von Komplexität und Verbesserung der Kostenschätzung [7].

Der Begriff „Product Line Engineering“ wurde von dem Software Engineering Institut (SEI) geprägt und verfeinerte das Konzept von Domain Engineering im Kontext eines Unternehmens:

“A software product line is a set of software-intensive systems that share a common, managed feature set satisfying a particular market segment’s specific needs or mission and that are developed from a common set of core assets in a prescribed way.” [15]

Die drei grundlegenden höchst iterativen Aktivitäten sind Core asset development”, „Product development“ und „Management“ wie in Abb. 33 ersichtlich.

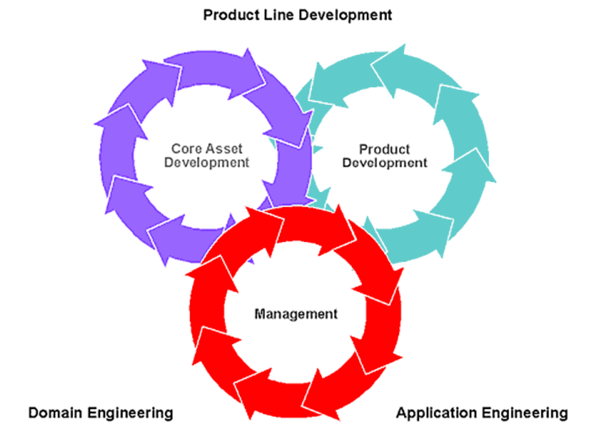


Abb. 3: Grundlegende Produktlinien Aktivitäten [17]

Core Asset Development – Entwicklung der Artefakte

Die wieder verwendbaren Artefakte bilden die Basis für die Entwicklung von Produkten. Core Assets sind Architekturen, wiederverwendbare Software Komponenten, Domänenmodelle, Anforderungen, Dokumentation, Spezifikationen, Performanzmodelle, Zeitpläne, Budget, Testpläne, Testfälle, Prozessbeschreibungen, etc. Die Architektur ist der Schlüssel für die Artefaktesammlung [17].

Als Input für die Entwicklung von Artefakten dienen Produktbedingungen in Form von Gemeinsamkeiten und Unterschiede, organisatorische Rahmenbedingungen für die Produktion und eine Produktionsstrategie. Als Output, neben den Artefakten, entsteht eine Produktlinienabgrenzung, die die möglichen Produkte beschreibt, sowie ein

Produktionsplan, wie Produkte mit Hilfe von den Artefakten erstellt werden können [15].

Die Ausarbeitung von Gemeinsamkeiten und Unterschiede wird häufig mit Featurediagrammen dokumentiert, wie sie in der Feature Oriented Domain Analysis (FODA) Methode angewendet wird. Dabei wird eine Hierarchie von Features aufgebaut. Es werden 3 Arten von Features unterschieden. Zwingende Features, die jedes System haben muss, alternative Features, bei denen nur ein Feature ausgewählt werden kann und optionale Features [5].

Artefakte können horizontale oder vertikale Wiederverwendung forcieren. Horizontale Artefakte sind generischer Art, nicht auf eine Domäne eingeschränkt und können daher produktlinien-übergreifend eingesetzt werden. Vertikale Artefakte sind mehr produktspezifisch [16].

Eine wichtige organisatorische Entscheidung betrifft, in welcher Form die Artefakte zur Verfügung gestellt werden. Werden sie von Grund auf neu implementiert, von Altsystemen extrahiert oder mit Commercial Off-The-Shelf (COTS) Komponenten zugekauft.

Diese Aktivität entspricht dem Domain Engineering Prozess, eingegrenzt in einem Unternehmenskontext. Diese Eingrenzung, mit der erweiterten Sicht über Code als Wiederverwendung hinaus, ermöglicht eine höherwertige Wiederverwendung.

Product Development - Produktentwicklung

Der Output der Artefaktentwicklung sowie konkrete Anforderungen dienen als Input für die Entwicklung der konkreten Produkte, die nach dem Produktionsplan entwickelt werden. Erfahrungen während der Entwicklung fließen wieder zurück in die Artefaktentwicklung. Die Produktentwicklung kann sehr stark variieren, abhängig von den Artefakten, dem Produktionsplan und der Organisation [15].

Diese Aktivität entspricht dem Applikations-entwicklungsprozess.

Management

Für die erfolgreiche Umsetzung von Product Line Engineering müssen die Artefakte- und Produkt-entwicklung geplant, koordiniert und überwacht werden. Dafür sind geeignete organisatorische Strukturen notwendig.

Es gibt 3 Vorgehensweisen, in welcher Form der Entwicklungsprozess gestartet wird [12]: proaktiv, reaktiv und extrahierend.

Bei dem proaktiven Ansatz werden zuerst die Artefakte implementiert und darauf aufbauend die Produktentwicklung eingeleitet. Dieser Ansatz ist bei stabilen Domänen geeignet, hat aber hohe Anfangskosten.

Beim reaktiven Ansatz werden die Artefakte nach Bedarf entwickelt und die Produkte nach herkömmlichen Verfahren erstellt. Anwendung findet dieser Ansatz bei Produkten, bei denen die Produktvariationen schwer vorherzusagen sind. Die Artefakte werden nachträglich aus den Produkten erstellt. Hohe Kosten können dann entstehen, wenn die Architektur angepasst werden muss.

Der extrahierende Ansatz wird eingesetzt, wenn schon Alt-Systeme bestehen. Aus diesen Systemen werden die Artefakte extrahiert und für neue Systemvarianten zu Verfügung gestellt. Eingesetzt wird diese Vorgehensweise bei Unternehmen, die von einer herkömmlichen Entwicklungsstrategie auf Produktlinien umstellen.

Bei der organisatorischen Strategie der Artefaktentwicklung werden der zentrale und verteilte Ansatz unterschieden [12].

Beim zentralen Ansatz gibt es eine organisatorische Einheit, die die Artefaktentwicklung betreibt. Der Vorteil ist, dass die Artefaktbasis einfacher koordiniert werden kann. Der Nachteil besteht in dem höheren Kommunikationsaufwand mit den Experten der Produktentwicklung.

Beim verteilten Ansatz übernehmen die einzelnen Projekte der Produktentwicklung auch die Artefaktentwicklung. Was der Vorteil der Koordination bei dem zentralen Ansatz ist, zeigt sich hier als Nachteil und umgekehrt.

SEI hat noch weitere Aktivitäten in Form von „Practice Areas“ definiert, die die 3 grundlegenden Aktivitäten unterstützen bzw. genauer definieren. Es sind 29 Areas, die in 3 Kategorien, Software Engineering, technisches Management und organisatorisches Management unterteilt sind [15].

Beispiele für Areas in der Kategorie Software Engineering sind Anforderungsanalyse, Architektur-Definition und –Evaluierung, Komponenten-entwicklung, COTS Einbindung, Testing. Im technischen Management die technische Planung, Risikomanagement, Konfigurationsmanagement, Prozessdefinition, Tool-Unterstützung. Im organisatorischen Management die Marktanalyse, Finanzierung, Planung, Organisationsstruktur, Risikomanagement, Training [15].

Wie diese Aktivitäten der Areas angewendet werden, wird in Patterns beschrieben. Patterns sind Lösungsansätze für wiederkehrende Probleme, wie sie schon in der technischen Applikationsentwicklung eingesetzt werden. Sie helfen bei der Einführung und

Umsetzung von Product Line Engineering in einem Unternehmen [15].

Es gibt einige Unternehmen, die Product Line Engineering erfolgreich eingesetzt haben. Darunter befinden sich Firmen wie Alcatel, Hewlett Packard, Philips, Boeing und Robert Bosch GmBh [15].

5. Domänenspezifische Modellierung (DSM)

Der am häufigsten eingesetzte Artefakttyp für die technische Umsetzung bei Produktlinien sind objekt-orientierte Software Komponenten, implementiert in einer objektorientierten „General Purpose“ (GL) Programmiersprache. Diese Komponenten werden wieder in GL Sprachen zu einem System zusammengefügt.

Ein weniger verbreiteter schwieriger Ansatz, aber viel versprechend im Grad der Wiederverwendung, sind modellorientierte Ansätze (MDE – Model Driven Engineering), die in direkten ausführbaren Code „kompiliert“ werden können bzw. mittels Code Generatoren in Applikationen transformiert werden. Vergleichbar mit jetzigen GL Sprachen, die mit Hilfe eines Compilers in Maschinencode übersetzt werden. Der Abstraktionslevel wird dadurch erhöht und man arbeitet näher an der Problemdefinition. Ein immenser Vorteil wäre, dass auch Domänen-Experten ohne Programmierkenntnisse Systeme erstellen könnten, da sie nur mit den Objekten und Aktionen ihrer Domäne arbeiten.

Modelle wurden schon früher als Dokumentation und Visualisierung für ein leichteres Verständnis eingesetzt. Beispiele für visuelle Modelle sind UML oder SASD. Von diesen Modellen aus kann auch eingeschränkt Code erzeugt werden. Diese Modelle basieren aber auf Konzepten von Programmiersprachen und „kennen“ die spezifischen Konzepte einer Produktfamilie nicht bzw. nur in Form von Namenskonventionen oder Stereotypen [18].

Auch Codegenerierung und „very high languages“ sind kein neues Konzept, aber durch hohe Anfangskosten und wenig unterstützende Tools haben sich diese Konzepte um 1990 noch nicht durchgesetzt [19]. Erst ab dem Jahr 2000 wurden konkrete Bestrebungen und Toolunterstützungen sichtbar.

Domain Specific Languages (DSL) - Domänen-spezifische Sprachen und Metamodelle

Einem Modell liegt eine domänenspezifische Sprache (DSML oder nur DSL) zugrunde. Sie bieten Notationen und Konstrukte in einer eingeschränkten Domäne. Sie bieten stärkere Ausdruckskraft und einfachere Handhabung als GL Sprachen [20]. Die Verwendung von domänenspezifischen Sprachen ist aber keineswegs neu. BNF, SQL, HTML, etc. sind Beispiele für domänenspezifische Sprachen. Die Erstellung von DSLs ist aber sehr aufwändig [21], [22].

Im Zusammenhang mit MDE ist eine DSL eine Menge von zusammenhängenden Modellen. Das Modell des Systems verwendet Konstrukte, die in einem Metamodell definiert sind. Die Konstrukte im Metamodell sind wiederum in einem Metametamodell spezifiziert. Das Metametamodell ist das letzte Glied in diesem „Metamodeling stack“ [20]. Es gibt genau 1 Meta- und Metametamodell, aber unterschiedliche Modelle, die dem Metamodell entsprechen. Abb. 4 zeigt die Zusammenhänge.

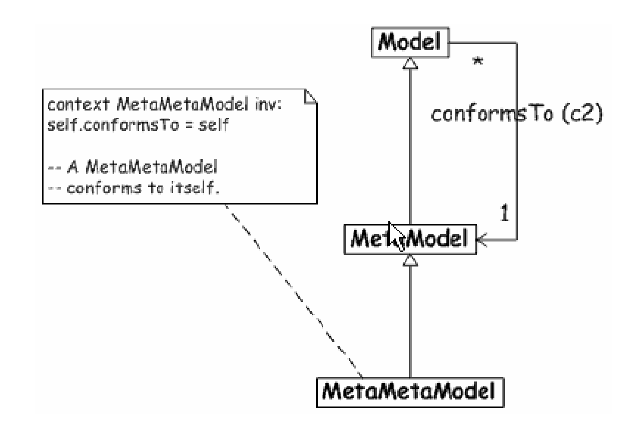


Abb. 4 Metamodeling Stack [20]

Als Beispiel sei hier XML erwähnt. Ein XML Dokument beinhaltet Elemente, die in einem XML Schema definiert sind, das dem Metamodell entspricht. Die Notation der Schemadefiniton ist wiederum in einem XML Metaschema definiert, das dem Metametamodell entspricht [20].

Da das Metamodell die Konzepte einer Domäne spezifiziert, beschreibt [20] aufgrund der Definition von Gruber T. R. das Metamodell als Ontologie.

Metamodeling Framework

Für die erfolgreiche Anwendung von DSM mit automatischer Codegenerierung müssen geeignete Werkzeuge in einer Entwicklungsumgebung vorhanden sein. [23] beschreibt ein Framework, in dem 3 Elemente notwendig sind.

1) ein Modellierungstool, das die domänen-spezifische Sprache unterstützt.
2) ein Code-Generator
3) domänenspezifisches Framework

Abb. 5: Framework für domänenspezifisches Modellieren zeigt die 3 Elemente auf 2 Ebenen. Auf der Spezifizierungsebene und auf der Verwendungsebene.

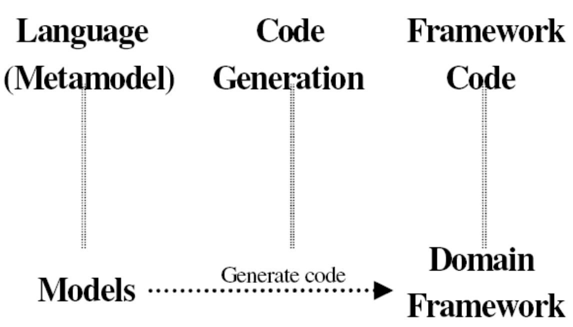


Abb. 5: Framework für domänenspezifisches Modellieren [23]

Die obere Ebene, der Spezifizierungsprozess, wird einmal durchgeführt bzw. bei Änderungen angepasst. Domänenexperten und Programmierer entwerfen das Metamodell und die domänenspezifische Sprache sowie den Code Generator. Das Metamodell ist die Implementation der domänenspezifischen Sprache und enthält die Konzepte und Regeln der Domäne. Der Framework Code beinhaltet technologiespezifischen Basiscode z.B. in Form von Komponenten [23].

Der untere Prozess zeigt die Verwendung der domänenspezifischen Sprache mit dem Code Generator. Dieser Prozess wird pro Produkt durchgeführt. Dabei geht der Code Generator durch das Modell und transformiert die konzeptuelle Struktur in Code basierend auf dem Framework Code. Dadurch, dass der Code erzeugt wurde, ergeben sich auch keine Implementierungsfehler [23].

Standards und Tools

Für die Umsetzung von Metamodeling müssen geeignete Tools und Standards entwickelt und definiert werden, um die Komplexität der Entwicklung zu unterstützen. Folgend eine kurze Auflistung einiger Standards und Tools.

Die Object Management Group (OMG), ein Konsortium aus verschiedenen Organisationen, verantwortlich für mehrere standardisierte Spezifikationen wie CORBA und UML hat im Bereich MDD die Model Driven Architecture spezifiziert [24].

Microsoft Factory-Ansatz versucht eine Plattform zu entwickeln, in dem eigene DSLs mit Hilfe von Templates definiert werden können [25].

Eclipse, eine offene Entwicklungsplattform, verfolgt das „Modeling" Projekt, das auf bestehende Eclipse Frameworks wie Eclipse Modeling Framework (EMF) und Graphical Modeling Framework (GMF) aufbaut [26].

MetaEdit+ von dem Unternehmen Metacase ist ein ausgereiftes Produkt für DSM. Unternehmen wie Nokia und EADS haben das Produkt schon erfolgreich eingesetzt [27].

6. Zusammenfassung

Ausgehend vom Artikel von Arango von 1989 waren die Konzepte von Domain Engineering im Zusammenhang mit Wiederverwendung in der Softwareentwicklung noch sehr jung, wurden aber zur Basis für die weiterführende Verfeinerung. Weiterentwicklungen gab es in Richtung Software Produktlinien, wo Domain Engineering im Umfeld eines Unternehmens betrieben wird und unternehmensweite Strategien entwickelt wurden. Code ist dabei für die Wiederverwendung nur ein möglicher Artefakt. Bei domänenspezifischer Modellierung wird das Wissen in Form von Modellen wieder verwendet, aus denen dann fertige Systeme generiert werden.

Der Trend der Wiederverwendung in der Softwareentwicklung geht hin zu flexiblen, anpassungsfähigen Plattformen, die verschiedene Werkzeuge für unterschiedliche Anwendungszwecke zu Verfügung stellen. Die gemeinsamen Entwicklungen und Standardisierungen unterschiedlicher Unternehmen und Organisationen im Bereich DSM und Wiederverwendung zeigen großes Interesse und Potenzial, die Softwareentwicklung weiter zu verbessern.

7. Referenzen

[1] Arango G. 1989. Domain analysis: from art form to engineering discipline. In *Proceedings of the 5th international Workshop on Software Specification and Design* (Pittsburgh, Pennsylvania, United States). IWSSD '89. ACM Press, New York, NY, 152-159

[2] McIlroy M. D., 1968. Mass produced software components. In Software Engineering; Report on a conference by the NATO Science Committee (Garmisch, Germany, Oct.). Naur, P., and Randell, B., Eds. NATO Scientific Affairs Division, Brussels, Belgium, pp. 138–150.

[3] Long J., IBM, Software reuse antipatterns, Source ACM SIGSOFT Software Engineering Notes , Volume 26 , Issue 4 , pp. 68 - 76 , July 2001

[4] Neighbors J. M., Software Construction Using Components, Doctoral Thesis, Department of Information and Computer Science, University of California, Irvine, 1980.

[5] Czarnecki K., Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models, Ph.D. Thesis, Department of Computer Science and Automation, Technical University of Ilmenau, October 1998

[6] Kean L., Domain Engineering and Domain Analysis, http://www.sei.cmu.edu/str/descriptions/deda.html, letzte Modifikation 1998, letzter Zugriff: 05.07.2007.

[7] Pohl K., Böckle G., van der Linden F., Software Product Line Engineering, Springer-Verlag Berlin Heidelberg, 2005

[8] Coplien J., Hoffman D., Weiss D., Commonality and Variability in Software Engineering, *IEEE Software*, vol. 15, no. 6, pp. 37-45, Nov/Dec, 1998

[9] Allen N. A., Shaffer C. A., Watson L. T. 2005. Building modeling tools that support verification, validation, and testing for the domain expert. In *Proceedings of the 37th Conference on Winter Simulation* (Orlando, Florida, December 04 - 07, 2005). Winter Simulation Conference. Winter Simulation Conference, 419-426

[10] Fowler M., Rice D., Foemmel M., Hieatt E., Mee R., Stafford R., Patterns of Enterprise Application Architecture, Addison Wesley, Pearson Education Inc, Boston, USA, November 2002

[11] Neighbors J. M., Bayfront Technologies, Inc., Draco: A Method for Engineering Reusable Software Systems, In Software Reusability: Vol. 1, Concepts and Models, T. J. Biggerstaff and A. J. Perlis, Eds. ACM Press, New York, NY, 295-319, May 1987

[12] Sugumaran V., Park S., Kang K. C., Software Product Line Engineering, Special Issue, Communications of the ACM, December 2006/Vol. 49, No. 12

[13] Mili A., Yacoub S., Addy E., Mili H., Toward an Engineering Discipline of Software Reuse, IEEE Software, vol. 16, no. 5, pp. 22-31, Sept/Oct, 1999

[14] Parnas D. L. On the Design and Development of Program Families, IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, pp. 1-9, März 1976.

[15] Northrop L. M., SEI's Software Product Line Tenets, IEEE Software, IEEE Software, vol. 19, no. 4, pp. 32-40, Jul/Aug, 2002

[16] Díaz-Herrera J. L., Domain Engineering, World Scientific, 2000

[17] http://www.sei.cmu.edu/productlines/framework.html, Letzter Zugriff: 12.07.2007

[18] Tolvanen J. P., Domänenspezifische Modellierungssprachen für Produktfamilien, ObjectSpektrum, August/September, 2001

[19] Krüger C. W., Software Reuse, ACM Computing Surveys, 24(2), pp. 131-183, June 1992

[20] Kurtev I., Bezivin J., Jouault F., Valduriez P., Model-based DSL Frameworks, In Companion To the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, Portland, Oregon, USA, October 22 - 26, 2006, OOPSLA '06, ACM Press, New York, NY

[21] Mernik M., Heering J., Sloane A.M., When and how to develop domain-specific languages, CWI Centrum voor Wiskunde en Informatica, Report SEN-E0309, December 2003

[22] Deursen A., Klint P., Visser J., Domain-specific languages: an annotated bibliography, ACM SIGPLAN Notices, Volume 35, Issue 6, June 2000

[23] Tolvanen J. P., Sprinkle J., Gray J., The 6th OOPSLA Workshop on Domain-Specific Modeling, ACM, OOPSLA'06 October 22–26, 2006

[24] http://www.omg.org/mda
Letzter Zugriff: 12.07.2007

[25] http://msdn2.microsoft.com/en-us/practices/bb190387.aspx
Letzter Zugriff: 12.07.2007

[26] http://www.eclipse.org/modeling/
Letzter Zugriff: 12.07.2007

[27] http://www.metacase.com/
Letzter Zugriff: 12.07.2007

Eine Einführung in Fuzzy Logik und seine geschichtliche Entwicklung
Sommersemester 2007

Nina Margarita Winkler
0360706
n2winkle@edu.uni-klu.ac.at

Abstract

Bis ins 20. Jahrhundert war die von Aristoteles begründete zweiwertige Logik die Basis von sämtlichen Ansätzen in Bezug auf Wissensrepräsentation. Die Problematik der bivalenten Logik ist jedoch das Unvermögen mit unsicheren und ungenauen Aussagen zufrieden stellend umzugehen.
Fuzzy Logik, die eine Erweiterung des klassischen Logiksystems ist, kann diese Schwierigkeit lösen. Im Gegensatz zur klassischen Logik lässt die Fuzzy Logik nicht nur zwei Werte wie wahr oder falsch zu, sondern ebenso Zwischenwerte, wodurch es ihr möglich ist die Unsicherheit und Ungenauigkeit von Daten effektiv handzuhaben.
Die Seminararbeit soll zunächst die grundlegenden Konzepte und Verfahren von Fuzzy Logik näher bringen und einen tieferen Einblick in die Konsequenzen dieser Kernidee verschaffen.

1. Vorwort

Die bivalente Logik ist ein mächtiges Instrument, auf dem viele Anwendungen aufbauen, aber mit ihrer Hilfe kann man die Welt nur vereinfacht darstellen. Die bestehende Welt ist meist viel zu umfassend, um sie mit binären Aussagen entsprechend darstellen zu können. So sind die meisten Wörter und Aussagen, die wir im alltäglichen Leben verwenden, nicht formell definierbar. Wenn man also komplexe Sachverhalte modellieren will, ist Fuzzy Logik das Mittel der Wahl.

Im 2. Kapitel werde ich näher in die Theorie und Praxis der Fuzzy Logik eingehen und versuchen grundlegende Verfahren anhand eines Beispiels zu erläutern. Im anschließenden Kapitel gehe ich auf die Entstehung und Entwicklung der Fuzzy Logik und auf ihre Anwendungen näher ein. Im 4. Kapitel werde ich einen Ausblick auf die Zukunft in diesem Forschungsgebiet geben und mit einer Schlussfolgerung meine Arbeit abschließen.

2. Einführung in Fuzzy Logik

In den Wissenschaften arbeitet man zur Darstellung von Systemen gewöhnlich mit mathematischen Modellen. Eine Problematik der Beschreibung von komplexen Systemen liegt darin, dass oft Vereinfachungen des Systems von Nöten sind, um ein geeignetes Modell zu entwerfen. Durch den Fortschritt in der Technik ist es uns möglich vielschichtige Systeme zu verwenden, aber mit dem Umfang des Systems werden auch die konzeptionellen Anforderungen voluminöser, die nötig sind, um zumindest eine gewisse Überschaubarkeit des Systems zu gewährleisten. Zudem gibt es auch Systeme, bei denen der Entwurf eines mathematischen Modells in einem angemessenen Zeitraum nicht möglich ist.

Bei Fuzzy Systemen erhält man durch das durchdachte Verwenden von nicht perfekten Informationen eine Reduktion der Komplexität des Systems bereits während der Modellierungsphase, wobei es selbsterklärlich ist, dass zu grob formulierte Beschreibungen die wirkliche Welt wiederum falsch abbilden [1].

2.1. Theorie der Fuzzy Mengen

In der klassischen Mengenlehre ist ein Element in einer Menge entweder enthalten oder nicht. Möchte man nun etwa eine Menge mit Elementen füllen, die alle eine angenehme Raumtemperatur ausweisen, so ist es notwendig eine sprunghafte Grenze zwischen den Elementen, die zur Menge gehören, und jenen, die nicht dazu gehören, zu ziehen.

Sei etwa die Menge M definiert durch *{x| 20 ≤ x ≤ 24}*, wie Abbildung 1 veranschaulicht.

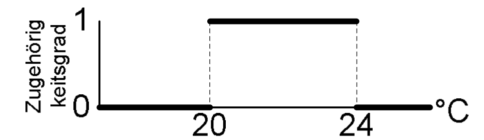


Abbildung 1: Klassische Menge

Während 20 Grad Celsius noch eine angenehme Raumtemperatur ist, gehört 19,9 Grad nicht mehr dieser Menge an und stellt somit eine unangenehme Raumtemperatur dar. Der Mensch empfindet jedoch keine solchen abrupten Grenzen, sondern eher fließende Abgrenzungen, welche man mit Fuzzy Mengen definieren kann, wie man in Abbildung 2 erkennen kann.

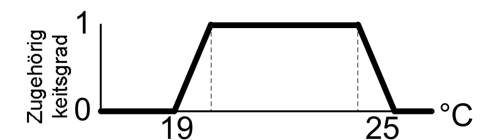


Abbildung 2: Fuzzy-Menge

Im Gegensatz zu der klassischen Mengenlehre gehört ein Element einer Fuzzy Menge dieser mit einem bestimmten Zugehörigkeitsgrad, üblicherweise im Intervall von null bis eins, an. So hat beispielsweise eine Temperatur von 19,9 Grad Celsius noch einen Zugehörigkeitsgrad von 0,9.

Eine Zugehörigkeitsfunktion benötigt man, um die Abstufungen einer Zugehörigkeit beschreiben zu können. In der vorhergehenden Abbildung haben war als die Zugehörigkeitsfunktion eine Trapezfunktion definiert.

Ebenso wie in der klassischen Mengenlehre gibt es auch bei Fuzzy Mengen viele Operationen, die man auf Elemente anwenden kann.

Genauso wie die bool'sche Algebra die Verknüpfungen der klassischen Mengenlehre zur bivalenten Logik herstellt, so ist dies auch bei der erweiterten Fuzzy Mengen-Theorie der Fall. Auch hier gibt es Übereinstimmungen, beispielsweise zwischen dem Durchschnitt und dem logischen UND, der Vereinigung und dem logischen ODER und weiters auch zwischen dem Zugehörigkeitsgrad eines Elementes in einer Fuzzy Menge und dem Wahrheitswertes einer Aussage in Fuzzy Logik.

Gleichfalls gibt es in der Fuzzy Logik auch Implikationen, also logische „Wenn - dann“ – Beziehungen, wodurch man Regeln definieren kann.

2.2. Fundamentale Verfahren

Um die primären Verfahren, die für die Entwicklung eines Systems mit Fuzzy Logik erforderlich sind, besser veranschaulich zu können, möchte ich ein simples Beispiel für ein leichteres Verständnis zu Hilfe nehmen. Hierbei orientiere ich mich im Besonderen an [2] und [1].

Beispiel: Ich will einen Regler für eine Klimaanlage erstellen. Von den Sensoren werden zwei scharfe[1] Messgrößen geliefert: die Temperatur im Raum und die Anzahl der Personen im Raum. Durch diese Informationen soll der Ausgangswert berechnet werden. Dieser Befehlswert soll den Prozentwert angeben, wie weit man das Ventil, welches für die Kühlung verantwortlich ist, öffnen soll.

Im Prinzip besteht ein Fuzzy Regler aus drei Teilen:

- § Fuzzifizierung
- § Inferenz
- § Defuzzifizierung

Abbildung 3 zeigt eine vereinfachte Sichtweise eines Fuzzy Reglers. Als Nächstes werde ich die Bestandteile und deren Zusammenwirken genauer erläutern und anschließend eine detaillierte Beschreibung der Entwicklung eines Fuzzy Systems und deren Funktionsweise geben.

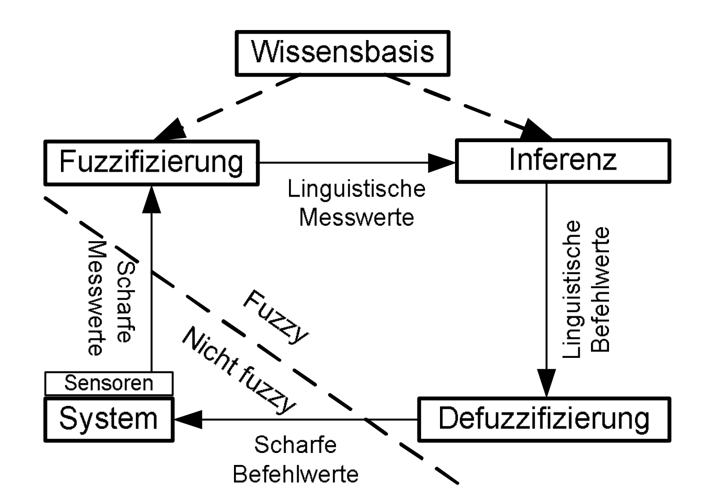


Abbildung 3: Architektur eines Fuzzy Reglers

[1] Ein scharfer Wert bezeichnet einen präzisen Wert, also gewöhnlich eine Zahl, während ein unscharfer beziehungsweise ein linguistischer Wert einen ungenauen Wert beschreibt, z.B.: eher heiß

Bestandteile eines Fuzzy Reglers und deren Verbindungen:

Bei den Messwerten des Reglers, die von den Sensoren geliefert werden, handelt es sich um scharfe Zahlenwerte. Diese müssen nun in die linguistischen Werte des Systems abgebildet werden. Dieser Schritt ist notwendig, da die Regeln auf unscharfen Werten operieren. Die Transformation der scharfen Messwerte in linguistische Messwerte geschieht anhand von Informationen aus der Wissensbasis (genauer: Datenbasis), die während der Modellierungsphase erstellt wurden. Diesen Vorgang nennt man Fuzzifizierung.

Als Nächstes werden mit Hilfe der nun linguistischen Messwerte die Regeln, entsprechend aufgelöst. Die Regeln sind ebenso Teil der Wissensbasis (genauer: Regelbasis) und werden ebenso während der Modellierungsphase konzipiert.

Anschließend müssen die unscharfen Befehlswerte wieder in scharfe Zahlen transformiert werden, da die Einstellgrößen für den Regler scharf sein müssen. Diesen Vorgang nennt man Defuzzifizierung.

Aufstellen der Wissensbasis: Datenbasis

Damit scharfe Zahlen in linguistische Werte abgebildet werden können, muss man die linguistischen Variablen und deren linguistischen Werte bestimmen.

Das Konzept einer linguistischen Variablen kommt aus unserer Alltagssprache. Dadurch werden Worte bezeichnet, deren Wertebereich durch unscharfe Mengen, so genannte linguistische Werte, abgedeckt werden [3]. Dieser Vorgang wird als Fuzzy Zerlegung oder Fuzzy Partition bezeichnet. Außerdem ist es möglich die Auswahl zu erweitern und zwar um Modifikatoren, Quantoren, Wahrscheinlichkeiten und Möglichkeiten. Dies ist für Systeme geeignet, deren verbale Beschreibung in vielerlei Hinsicht sehr unscharf ist, so dass dieses Wissen nur so korrekt formuliert werden kann. Dies soll am Beispiel der linguistischen Variable Temperatur veranschaulicht werden. Weitere Informationen über die Wissensrepräsentation von unscharfen Aussagen kann man in [4] und [5] erhalten.

Linguistische Variable:	Temperatur
Linguistische Werte	Primärterm: heiß
	Antonym: kalt
Modifikatoren	Sehr, wenig, nicht, kaum
Quantoren	Viele, wenige, einige, alle
Möglichkeiten	vielleicht, eher unmöglich
Wahrscheinlichkeiten	Meist, manchmal, oft

Die Festlegung der linguistischen Variablen und ihren Werten findet man bei einfachen Problemen praktisch intuitiv. Bei komplexen Prozessen ergeben sich diese jedoch nicht so leicht. Wählt der Entwickler die linguistischen Variablen nicht realitätsgetreu wird es wahrscheinlich nötig sein die Datenbasis wiederholt zu konzipieren. Das ist etwa der Fall, wenn sich später herausstellt, dass mit Hilfe der Messwerte keine vernünftige Regelung möglich ist. Ebenso verhält es sich mit den Wertebereichen der linguistischen Variablen. Oft zeigt die Natur Grenzen auf, wie beispielsweise kann ein Winkel nur Werte zwischen 0 und 360 Grad annehmen. Allerdings gibt es auch Größen, die keine klaren Wertgrenzen besitzen und der Entwickler des Systems selbst zu entscheiden hat, wo es vernünftig ist diese zu setzen.

Nach der Bestimmung des Wertebereichs muss dieser nun in die linguistischen Werte partitioniert werden. Außerdem muss die Form der Zugehörigkeitsfunktion für die linguistischen Werte festgelegt werden. Die Form der Funktion liegt ebenso in der Verantwortung des Entwicklers. Meist werden jedoch simple Zugehörigkeitsfunktionen gewählt, wie beispielsweise die Dreiecks- oder Trapezfunktion oder auch die Gauß'sche Glockenkurve. Weitere Informationen über Zugehörigkeitsfunktionen und das Aufstellen der Datenbasis kann man in [1], [2], [6] und [7] finden.

Beispiel: Erstellen der Datenbasis

Als Eingangswerte erhalte ich die Größen Temperatur und Personenanzahl und weiters ist der Ausgangswert die Stellung des Ventils zur Kühlung. Mit diesem Wissen kann ich die linguistischen Variablen inklusive deren Werten wie folgt wählen:

§ Temperatur (kalt, angenehm, heiß)
§ Personen(wenig, mittel, viele)
§ Ventil(zu, wenig offen, viel offen, offen)

Deren Zugehörigkeitsfunktionen bestimme ich wie in den nächsten Abbildungen dargestellt.

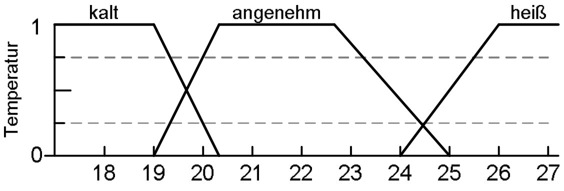


Abbildung 4: Zugehörigkeitsfunktion der linguistischen Variable Temperatur

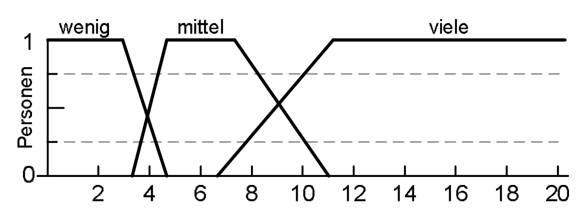


Abbildung 5: Zugehörigkeitsfunktion der linguistischen Variable Personen

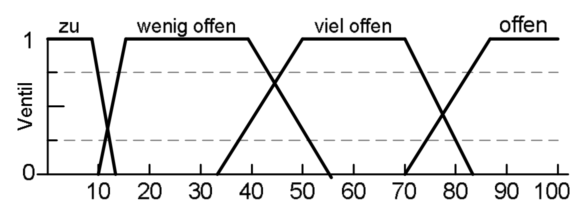


Abbildung 6: Zugehörigkeitsfunktion der linguistischen Variable Ventil

Aufstellen der Wissensbasis: Regelbasis

Nach der Bestimmung der linguistischen Variablen, deren Werte und deren Zugehörigkeitsfunktionen werden nun die Regeln für das System mit Hilfe der linguistischen Terme aufgestellt. Es gibt verschiedene Methoden diese Regeln zu erhalten. Beispielsweise wird oft ein Experte des jeweiligen Bereichs zu Rate gezogen, welcher dann seine Handlungen in linguistischen Regeln formuliert. Die Aufgabe des Entwicklers ist es diese Regeln in Implikationen mit linguistischen Termen zu transformieren.

Außerdem ist für die Art der Auswertung der Regeln eine Methode festzulegen. Wir verwenden die einfache und durchaus erfolgreiche Min-Max-Inferenz-Methode. Diese werde ich beim Auswerten näher beschreiben.

Beispiel: Aufstellen der Regelbasis

R1: WENN Temperatur=kalt DANN Ventil=zu

R2: WENN Temperatur=heiß DANN Ventil=offen

R3: WENN Temperatur=angenehm UND Personen =wenig DANN Ventil=zu

R4: WENN Temperatur =angenehm UND Personen =mittel DANN Ventil=wenig offen

R5: WENN Temperatur =angenehm UND Personen =viele DANN Ventil=viel offen

Auswahl der Defuzzifizierungsstrategie

Als letzte Aufgabe des Entwicklers in der Modellierungsphase ist eine geeignete Defuzzifizierungsstrategie zu wählen.

Eine sehr beliebte Strategie ist die so genannte Schwerpunktmethode [7]. Diese ermittelt den scharfen Wert als Flächenschwerpunkt. Diese Methode ist sehr zuverlässig, jedoch ist die Schwerpunktberechnung, speziell wenn die Zugehörigkeitsfunktion aus Kurven besteht, recht mühsam und erfordert dementsprechend auch viel Speicherplatz und Zeit. Ein Grund warum man etwa bei Systemen, in denen es auf Echtzeit ankommt, eher andere Methoden vorzieht.

Eine andere Strategie wäre die Maximum-Methode [6]. Diese gewinnt den scharfen Wert aus dem Maximum der Zugehörigkeitsfunktion. Dies führt zu Problemen, wenn es mehrere Maxima gibt, was oft der Fall ist. Wir entscheiden uns aus Einfachheitsgründen für die Links-Maximum-Methode, die das linkste Maximum als scharfen Wert wählt.

Beispiel: Berechnung des Befehlwerts

Nun soll der scharfe Befehlwert, abhängig von den zwei Eingangswerten berechnet werden. Es wird eine Temperatur von 20 Grad Celsius und weiters eine Anzahl von zehn Personen gemessen.

Nun werden die Aktivierungsgrade dieser Messwerte bestimmt:

Variable	Wert	Grad	Term
Temperatur	20° Celsius	0,75	angenehm
		0,25	kalt
Personen	10	0,25	mittel
		0,75	viele

Die Regeln werden nun mit Hilfe der Min-Max-Inferenz-Methode aufgelöst. Beachte: Min-Max-Inferenz-Methode: bei UND wird der kleinere Wert genommen (und der größere Wert bei ODER). Lösen wir etwa die Regel R4 auf. Die Temperatur ist mit Grad 0,75 angenehm und mit Grad 0,25 mittel. Da in der Regel eine UND-Verknüpfung ist, nimmt man den kleineren Wert als Grad für diese Regel. Diesen Vorgang der Auflösung der Regeln nennt man auch approximatives Schließen.

Regel	Wenn-Teil	Grad	Dann-Teil
R1	0,25	0,25	Ventil=zu
R2	0	0	Ventil=offen
R3	0,75 AND 0	0	Ventil=zu
R4	0,75 AND 0,25	0,25	Ventil=wenig offen
R5	0,75 AND 0,75	0,75	Ventil=viel offen

Als nächstes werden nach diesen Informationen die jeweiligen Bereiche in der Zugehörigkeitsfunktion der linguistischen Variable Ventil ausgefüllt, wie in Abbildung 7 abgebildet.

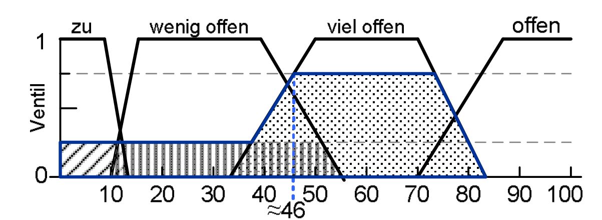


Abbildung 7: Links – Maximum - Methode

Beispielsweise wird der linguistische Wert „zu“ bis zu Grad 0,25 (aufgrund Regel R1) ausgefüllt. Dies wiederholt man mit allen Termen.

Um einen scharfen Wert zu erhalten, wenden wir die Links-Maximum-Methode an und erhalten 46 % als Befehlwert für die Ventilstellung.

Gewöhnlicherweise ist das Verhalten des Reglers nach der ersten Darstellung noch nicht ganz optimal. Es sind kleine Verbesserungsmaßnahmen notwendig, wie beispielsweise die erneute Spezifikation der Lage oder der Zugehörigkeitsfunktion der Fuzzy Mengen, eine neue (feinere) Partition des Wertebereichs oder eine neue Formulierung der Regeln. Bei schlechter Wahl der Mess- und Befehlwerte, kann es jedoch auch sinnvoll sein, einen gänzlichen Neuanfang zu starten.

3. Geschichtlicher Hintergrund (Auswirkungen der Kernidee)

3.1. Früher Anfang von Fuzzy Logik

Als vor über 2300 Jahren Aristoteles und seine Anhänger ihre Hypothesen über Logik entwickelten, kamen sie auf das so genannte Gesetz des ausgeschlossenen Dritten. Dieses Gesetz besagt, dass jede Aussage entweder falsch oder wahr sein muss. Dieses bivalente Logiksystem setzte sich, wohl auch aufgrund seiner Klarheit durch, obwohl es ebenso damals bereits Ansätze für mehrwertige Logik gab. Etwa sprach Platon der Lehrer von Aristoteles von einer dritten Region zwischen wahr und falsch.

Im 19. Jahrhundert entwickelte George Boole die nach ihm benannte Algebra und Mengenlehre, wodurch es endlich möglich wurde jenes Logiksystem auch mathematisch zu behandeln. Bereits Anfang des 20. Jahrhunderts schlug Jan Lukasiewicz jedoch ein trivalentes Logiksystem, mit den Werten wahr, möglich und falsch, vor, welches jedoch kein großes Interesse erfahren durfte.

1965 veröffentlichte Lotfi Zadeh die erste Arbeit [8] in Bezug auf Fuzzy Logik. Es ist dabei unbedingt zu erwähnen, dass er sich innerhalb eines kurzen Zeitraums von der präzisen Regeltechnik zur Fuzzy Logik zuwandte, die Unklarheit zulässt [9]. Er bezeichnete den Ursprung seiner Idee später [10] als das Prinzip der Unvereinbarkeit (engl.: principle of incompatibility):

As the complexity of a system increases, it becomes more difficult and eventually impossible to make a precise statement about its behavior, eventually arriving at a point of complexity where the fuzzy logic method born in humans is the only way to get at the problem.

Dieses Prinzip besagt, dass hohe Präzision nicht kompatibel ist mit hoher Komplexität. Ähnlich dem menschlichen Denken ermöglicht Fuzzy Logik dem Computer unscharfe Informationen zu verwerten.

Besonders amerikanische aber auch viele europäische Wissenschaftler und Forscher ignorierten anfangs diese neue Theorie. Auf der einen Seite lag dies daran, dass der Name irreführend gewählt war, was zur Folge hatte, dass viele Wissenschaftler dachten Fuzzy Logik würde die Präzision aus der Wissenschaft nehmen. Auf der anderen Seite war diese Theorie auch sehr provokant. Viele Wissenschaftler dachten, Fuzzy Logik würde unter anderem besagen, dass die konventionelle bivalente Logik, auf der praktisch ein Großteil der Wissenschaft aufbaut, falsch ist.

Der Universitätskollege und auch guter Freund William Kahan von Lotfi Zadeh hielt absolut nichts von der Fuzzy Logik Theorie, was er 1975, ein Jahr nach der Implementierung der ersten Anwendung, sehr provokant aussprach:

“Fuzzy theory is wrong, wrong, and pernicious. I can not think of any problem that could not be solved better by ordinary logic.”

Jedoch wurde Fuzzy Logik von den Wissenschaftlern aus Asien, allen voran Japan und China, akzeptiert, was definitiv zu einem Teil auf die damalige Vormachtstellung der Japaner im Bereich der Halbleitertechnologie und Mikrorechner zurückzuführen ist. Ein weiteres Argument ist jedoch auch die asiatische Betrachtungsweise, die im Gegenteil zur westlichen Tradition, in der Unschärfe nichts Verwerfliches sieht [1], [9].

3.2. Weitere Entwicklung von Fuzzy Logik

In der Anfangszeit von Fuzzy Logik behandelten die erschienenen Publikationen in der Mehrheit die theoretische Untersuchung von Erfolg versprechenden Verfahren. Ein Resümee der bis 1978 veröffentlichten Artikel zum Thema kann man in [11] finden.

Von Anfang der 70er Jahre bis in die frühen 80er Jahre wurdem die ersten Anwendungen, überwiegend

im Bereich der Regelungstechnik, implementiert. Ich werde nun eine Auswahl von ihnen vorstellen [2].

Mamdanis kleine Dampfmaschine

L.A. Zadeh publizierte erste Arbeiten im Hinblick auf das Konzept zur Regelung mit Hilfe von Fuzzy Logik in den frühern 70ern. Die erste Implementation (im Labormaßstab) erstellten E.H. Mamdani, ein Professor an der Universität von London, und sein Student Seto Assilian im Jahr 1974 [12]. Es handelte sich um einen Fuzzy-Regler, der bei einer Dampfmaschine für die Steuerung der Ventilstellung zur Dampfzufuhr und ebenso für die Wärmezufuhr zur Dampferzeugung verantwortlich war.

Linkman

Ebenfalls wie die erste Anwendung kam auch die erste größere, industrielle Realisierung eines Fuzzy-Reglers aus Europa. Die aus Dänemark stammende Applikation mit dem Namen Linkman, dessen Aufgabe es war die Zementbrennprozesse zu steuern, wurde 1982 implementiert.

Subway

Nach acht Jahren strenger Untersuchungen und zahlreichen Tests wurde 1986 in Sendai eine U-Bahn in Betrieb genommen, die vollständig mit Fuzzy - Reglern ausgestattet ist und vollautomatisch fährt. Durch diese Anwendung konnte die Fuzzy Theorie demonstrieren, dass man sie auch in Situationen einsetzen kann, in denen Sicherheit von enormer Bedeutung ist.

Durch den erfolgreichen Betrieb der U-Bahn wurde die Fuzzy Logik in Japan sehr beliebt und man erkannte ihre Möglichkeiten. Folglich wurden zahlreiche Institutionen und Organisationen gegründet, die sich mit Fuzzy Logik beschäftigen. Etwa 1987 das IFSA(International Fuzzy Systems Association) oder 1989 das japanische LIFE(Laboratory for International Fuzzy Engineering Research)-Institut, das vom japanischen Ministerium für Handel und Industrie und von 48 Unternehmen gefördert wird.

In den späten 80er Jahren erscheinen viele Arbeiten in Japan, die zwar zu einem Großteil keine neuen Konzepte in der Fuzzy Logik Theorie aufwerfen sollten, aber über neue Anwendungen in der Regelungstechnik handelten. Durch die vielen Produkte, vor allem Haushaltsgeräte, die die Fuzzy Logik Verfahren implementierten, wurde in Japan ein regelrechter Fuzzy Boom ausgelöst, der seine Spitze 1990 erreichen sollte. In diesem Jahr wurde fuzzy zum Wort des Jahres in Japan gewählt. Des Weiteren entwickelten japanischen Wissenschaftler Hardware Lösungen, um die Leistung von Fuzzy-Systemen zu erhöhen. Durch den immensen Erfolg in Japan wurde die Wirtschaft auch in Europa und in Nordamerika hellhörig [1], [9].

3.3. 90er Jahre bis heute

Bis in den Anfang der 90er Jahre war das hauptsächliche Anwendungsgebiet von Fuzzy Logik die Regelung zum einen in industriellen Systemen und zum anderen in Konsumgütern. Doch das sollte sich ändern. Man „entdeckte" Fuzzy Logik ebenso für die Automatisierungstechnik, für die Unterhaltungselektronik und für die Steuerungselektronik. So eroberte diese Theorie viele unterschiedliche Bereiche, deren Anwendungen mit Hilfe von Fuzzy Logik eine bessere Qualität und Leistung aufwiesen.

Durch die immer höhere Komplexität von Fuzzy Logik Systemen wurde es immer wichtiger Richtlinien zur Entwicklung von Systemen zu erstellen. Dabei will ich die Entwicklungsmethodik (engl.: development methodology) vorstellen. Sie umfasst fünf Phasen:

Design:

1. Spezifizierung der linguistischen Variablen
2. Definition der Struktur der Schlussfolgerungen
3. Formalierung der Fuzzy - Regeln

Debugging:

4. Off-line Analyse, Testen, Verifikation
5. On-line Optimierung

Weitere Informationen und Erklärungen zu diesem Thema findet man in [13].

Seit Beginn der 90er Jahre wurde intensiver nach Möglichkeiten geforscht, die das Konzept von Fuzzy Logik mit anderen Theorien verbinden. Zwei der meines Erachtens interessantesten Konzepte werde ich hier kurz erläutern und sie über deren jetzigen Stellenwert informieren.

Bei einem Hybriden System, welches sich besonders gut hat etablieren können, handelt es sich um Neuro-Fuzzy Systeme. Darunter versteht man eine Symbiose von den beiden Konzepten neuronale Netze und Fuzzy Logik. Künstliche neuronale Netze bestehen aus vielen kleinen Neuronen, wobei Informationen durch die Verbindungen zwischen den Neuronen erfolgen, ähnlich den Vorgängen, die sich im menschlichen Gehirn abspielen. Ein großer Vorteil von neuronalen Netzen ist die hohe Lernfähigkeit. Durch die Kombination dieser zwei Theorien werden ihre beiden Schwachstellen ausgemerzt. Die Neuronalen Netze können einerseits durch Fuzzy Logik eine Wissensrepresentation erzeugen, die die Sachverhalte

des Systems wirklichkeitstreuer beschreibt. Andererseits wird das größte Manko der Fuzzy Logik ausgemerzt: Seine Unfähigkeit aus Fehlern zu lernen [6].

Ein weiterer Ansatz der sich mit Fuzzy Logik gut kombinieren lässt, sind die so genannten evolutionären Algorithmen. Diese nehmen sich die Prinzipien natürlicher Evolution und der Genetik zum Vorbild. Nach [6] kann man die Evolution als einen Lernprozess, der sich auf einfache Prinzipien stützt, verstehen. Die wichtigsten dieser Strategien sind

§ Crossover (Rekombination),
§ Mutation,
§ Inversion.

Da durch diese Algorithmen die bestmöglichste Lösung für ein Problem gefunden wird, kann man bei Fuzzy Systemen evolutionäre Algorithmen als Optimierungshilfe für die Wissensbasis verwenden. So kann etwa die Partitionierung von den linguistischen Variablen von einem evolutionären Algorithmus erledigt werden. Außerdem können die Regeln dadurch optimiert werden. Ein Manko ist, dass es nicht sicher ist, dass man mit dieser Methode in einer vernünftigen Zeitspanne auf ein optimales Ergebnis kommt. Eine sehr gute Einführung in evolutionäre Algorithmen in Verbindung mit Fuzzy Logik kann man in [6] finden.

Den Sammelbegriff von Fuzzy Logik, evolutionäre Algorithmen und künstliche neuronale Netze sind unter Soft-Computing bekannt.

Weiters sind dies Konzepte, die häufig im Bereich der künstlichen Intelligenz angewendet werden. So ist es mit Neuro-Fuzzy Systemen möglich, dass mobile Roboter lernen können, wo sich welche Hindernisse befinden, um Kollisionen aus den Weg zu gehen. Zudem können sich dadurch mobile Roboter auch in sich ändernden Umgebungen zurecht finden.

4. Ausblick

Einen Anstieg des Gebrauchs von Fuzzy Logik in jenen Anwendungsgebieten, in denen sich diese Theorie bereits in den Anfängen etabliert hat, ist anzunehmen. Dies wird besonders in der Regelungstechnik und bei Expertensystemen der Fall sein [14]. Die Qualität der Applikationen, die auf dem Fuzzy Logik Konzept basieren, wird zudem immer besser.

Ein Anwendungsgebiet von Fuzzy Logik, welches zwar nicht neu aber in Zukunft noch viel mehr an Bedeutung gewinnen wird, ist die künstliche Intelligenz. Neben Expertensystemen gibt es hier bereits Anwendungen zur Mustererkennung und in der Robotik. Des Weiteren wird auch besonders der Bereich des Sprachverstehens (engl.: natural language processing) von Fuzzy Logik geprägt. Da in der künstlichen Intelligenz das menschliche Denken als Vorbild fungiert, ist es leicht erklärlich, dass Fuzzy Logik schnell Einzug in diesen Bereich gefunden hat. Fuzzy Logik hat bereits einen großen Einfluss auf viele Bereiche der künstlichen Intelligenz. Da wir am Beginn des Zeitalters der Intelligenz von Maschinen stehen, werden noch viele Anwendungen in diesem Forschungsgebiet folgen.

Das Sprachverstehen etwa ist bisher eine relativ schwerer Forschungsbereich gewesen. Die Forschung versucht auf der einen Seite das „Verstehen" einer natürliche Sprache zu ermöglichen und deren Bedeutung in eine formalere Sprache abzubilden und andererseits versucht sie automatisch eine natürliche Sprache auf Grund von formalen Informationen zu generieren. Damit dies überhaupt möglich ist, ist es notwendig, dass ein Computerprogramm den Sinn von Texten versteht. Dies und auch die Wissensrepräsentation sind die Hauptprobleme in diesem Bereich, welche durch Fuzzy Logik gelöst werden.

Ein recht neues Forschungsgebiet ist das Internet. Der Trend geht in Richtung intelligente Suchmaschinen. Das semantische Netz benötigt eine Wissensrepräsentation und auch Schlussfolgerungen, welche mit verbalen Wörtern, also ungenauen, unscharfen Werten umzugehen weiß. Momentan arbeiten bereits Spamfilter mit einem Steuerungssystem, basierend auf der Fuzzy Logik Theorie.

Zadeh behauptet, dass es bereits in relativ naher Zukunft möglich sein wird, mit Hilfe von Fuzzy Logik, in einer natürlichen Sprache Befehle an Computerprogramme beziehungsweise insbesondere an Internetanwendungen zu richten. Die Programme sollen dann die Befehle des Benutzers „verstehen" und befolgen.

Insgesamt sind Zadeh und weitere Wissenschaftler, die im Gebiet der Fuzzy Logik forschen, der Meinung, dass in naher Zukunft das Operieren mit Fuzzy Logik ein großer Bestandteil unseres Alltagsleben wird. Wobei der Benutzer bei der Anwendung wahrscheinlich nicht erfährt, welche Theorien und Technologien dafür erforderlich sind.

5. Schlussfolgerung

Trotz herber Kritik in den Anfangsjahren (Rudolf Kalman, William Kahan) wurde vor allem an Universitäten theoretisch geforscht. Ebenso wurden die ersten Prototypen mit Fuzzy Logik von

Universitätsprofessoren implementiert. Erst als es der Industrie als lukrativ erschien, hat diese auch in Forschungen investiert. Darin kann man den Wert der Forschung an Universitäten erkennen. Erst die Erfolge in der Industrie haben auch Wissenschaftler in Amerika und Europa von den Vorteilen der Fuzzy-Logik überzeugt. Allerdings bemangelt Zadeh das Fehlen von größeren Zuwendungen für Forschung der Regierungen in Europa. So erhalten die Universitäten in Nordamerika viel höhere finanzielle Förderungen als es in den meisten europäischen Ländern der Fall ist.

Generell haben Fuzzy-Systeme eine schnellere und feinere Rückmeldungen als konventionelle Systeme. Weitere Charakteristiken sind Energiesparsamkeit, Reduzierung der Instandhaltungskosten und die Ausdehnung der Maschinenzeit. Die Beschreibung der Regeln in Fuzzy-Systemen in den meisten Fällen simpler beziehungsweise sind oft auch weniger Regeln notwendig. Dadurch gelingt die Ausführung dieser Systeme schneller als bei konventionellen Systemen. Des Weiteren erreichen Fuzzy-Systeme meist eine hohe Robustheit und eine ganzheitliche Kostenreduzierung. In Gesamtheit führen alle diese Eigenschaften zu einer besseren Leistung des Systems [2].

Trotz der Vorzüge von Fuzzy-Logik ist die klassische Logik ein mächtiges Werkzeug und eignet sich vor allem zur Lösung von simplen Problemen, während Fuzzy-Logik komplexe Probleme gut lösen kann oder auch Anwendungen unterstützt, die unsichere Informationen verarbeiten. In diesem Sinne stehen Fuzzy Logik und klassische Logik nicht im Konkurrenzkampf zueinander. Gleichfalls wie die Relativitätstheorie von Einstein das Gravitationsgesetz von Newton erweitert, ist die Fuzzy Logik als eine Erweiterung des bivalenten Systems zu sehen.

6. Referenzen

[1] R. Kruse, J. Gebhardt, F. Klawonn, “Fuzzy-Systeme”, *B. G. Teubner, Stuttgart,* 1993

[2] T. Munakata, Y. Tani, “Fuzzy Systems: An Overview”, *Communications of the ACM*, Vol 37 (3), März 1994, pp. 68-76

[3] L.A. Zadeh, “Fuzzy Logic”, *IEEE Computer Society*, Vol 21 (4), April 1988, pp. 83-93.

[4] L.A. Zadeh, “Commonsense Knowledge Representation Based on Fuzzy Logic”, *IEEE Computer*, Vol 16 (10), Oktober 1983, pp. 61-65.

[5] L.A. Zadeh, “Knowledge Representation in Fuzzy Logic”, *IEEE Trans. on Knowledge and Data Engineering*, Vol 1 (1), März 1989, pp. 89-100.

[6] A. Grauel, “Fuzzy-Logik – Einführung in die Grundlagen mit Anwendungen”, *BI-Wissenschaftsverlag, Mannheim*, 1995

[7] H.-H. Bothe, “Fuzzy Logic – Einführung in Theorie und Anwendungen”, *Springer-Verlag, Berlin,* 1993

[8] L.A. Zadeh, “Fuzzy Sets”, *Information and Control*, Vol 8 (3), Juni 1965, pp. 338-353

[9] K. Tanaka, “An Introduction to Fuzzy Logic for Practical Applications”, *Rassel Inc., Japan,* 1991

[10] L.A. Zadeh, “Outline of a New Approach to the Analysis of Complex Systems and Decision Processes”, *IEEE Trans. on Systems, Man and Cybernetics*, Vol SMC-3, Januar 1973, pp. 28-44.

[11] D. Dubois, H. Prade, “Fuzzy Sets and Systems – Theory and Applications”, *Acadamic Press, New York,* 1980

[12] E.H. Mamdani, S. Assilian, “An Experiment in Linguistic Synthesis with a Fuzzy Logic Controller”, *International Journal of Man-Machine Studies*, Vol 7 (1), Januar 1975, pp. 1-15

[13] C. Von Altrock, “Fuzzy Logic and Neuro – Fuzzy Applications Explained”, *Prentice Hall, New York,* 1995

[14] H.-J. Zimmermann, “Fuzzy Set Theory – And its Applications ”, *Kluwer Acadamic Publishers Boston/ Dordrecht/ London,* 2001

Computing with Words verstehen

Stefan Labitzke
0560486
slabitzk@edu.uni-klu.ac.at

Abstract

Das Ausgangspaper „Fuzzy Logic = Computing with Words“ von L.A.Zadeh behandelt die Zusammenhänge zwischen Computing with Words und der Fuzzy-Logik. Dabei wird auf die wesentlichen Vorteile von Computing with Words eingegangen.

Der vorliegende Artikel befasst sich einleitend mit den Wurzeln von Fuzzy-Logik. Das zu bearbeitende Paper ist 1996, also in der Aufschwungszeit von Fuzzy-Logik entstanden. Daher stellt es eine Schnittstelle zwischen der Theorie von Fuzzy-Logik und dem Einsatz dieser Methode dar.

Nachdem also der Basisartikel in den Grundzügen näher gebracht wurde, werden die Einsatzgebiete in den unterschiedlichen Bereichen, die Prämissen, sowie wichtige Werkzeuge zur Anwendung von Fuzzy-Logik erläutert.

1. Einleitung

Fuzzy-Logik wird oft fälschlicherweise als eine unscharfe bzw. verschwommene Theorie verstanden. Vielmehr erlaubt sie es, der strikten Bool'schen Logik Zwischenwerte einzufügen. Damit kann das menschliche Denken besser abgebildet werden. Die Fuzzy-Logik bildet den Grundstein für Computing with Words. Es werden hierbei die Grundmechanismen bereitgestellt, welche für Computing with Words benötigt werden.

Normalerweise versteht man unter Computation, die Verwendung und Manipulation von Zahlen und Symbolen. Im Gegensatz dazu verwenden Menschen bei Berechnungen und Schlussfolgerungen Wörter, welche logischerweise aufgrund von unterschiedlichen Auslegungen dazu neigen unpräzise, also fuzzy, zu sein. Man interpretiert also das Ergebnis und fügt somit eine bestimmte Unschärfe hinzu, mit der Konsequenz, dass man dadurch ein besseres Verständnis der Berechnung aufgrund der realitätsnahen Sichtweise der Lösung schafft.

Das Gleiche gilt bei dem Einsatz von Wörtern bei Computing with Words. Computing with Words verbindet natürlichsprachliche Wörter und Fuzzy-Variablen. Genau diese Verbindung macht Computing with Words zu einer eigenen Methode, welche unter genau den beiden oben genannten Gesichtspunkten eingesetzt wird:

- Die zur Verfügung stehende Information beinhaltet keine präzise und konkrete Darstellung, also kann sie nicht direkt als Wert angesehen werden.

- Es ist eine Unschärfe bei der Berechnung erlaubt, welche jedoch die Vorteile von besserer Nähe zur Realität, Nachvollziehbarkeit, Robustheit und geringerer Kosten mit sich bringt.

Oft kann nicht mit wirklichen Zahlen gerechnet werden. Es gibt einfach keinen Messwert für eine Eingabe, die ein System mit normaler Logik verarbeiten kann. So kann man bei einer Waschmaschine nicht den Verschmutzungsgrad in Prozent angeben. Dieser wird jedoch benötigt, um eine optimale Waschmittelmenge in den Waschmittelkorb zu geben. Daher greift das System auf eine unscharfe Menge zurück- wie wir später noch sehen werden.

Um dem Leser das Ganze näher zu bringen wird schließlich ein Tool vorgestellt, welches zur Simulation von Fuzzy-Logik eingesetzt wird. Dies wird durchgeführt, damit das zuvor in diesem Artikel vorgestellte Thema auch durch direktes Anwenden näher gebracht wird.

Zunächst jedoch einige Grundbegriffe sowie Definitionen, auf welche der Artikel im Weiteren aufbaut [1].

2. Grundbegriffe und Definitionen

2.1. Fuzzy-Logik

Die nachfolgenden Definitionen bilden die Basis für Computing with Words. Sie wurden im Wesentlich in [2] erklärt.

Ein Fuzzy-Set wird durch eine Membership-Funktion definiert. Dabei wird in mehrere Klassen unterschieden, welche jede durch eine Funktion f(x) beschrieben wird [9] [10]. Diese Funktion liefert für einen konkreten Wert im Intervall [0-1], die Zugehörigkeit dieses Wertes zur jeweiligen Klasse.

Ein kurzes Beispiel wird dies erläutern:

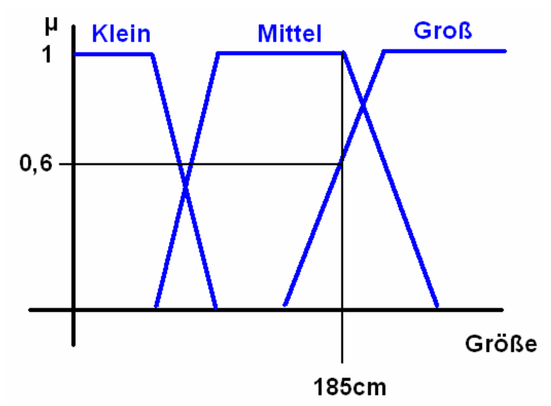


Abb. 1: Membership-Funktionen

In Abbildung 1 werden 3 Membership-Funktionen mit den 3 Klassen *Klein*, *Mittel* und *Groß* auf die Fuzzy-Variable „Größe" dargestellt. Zieht man nun einen Wert von 185cm für die Größe heran, so erhält man aus dem Diagramm die Fuzzy-Werte 0 für *Klein*, 1 für *Mittel* und 0,6 für *Groß*.

Als nächsten Schritt gehen wir nun näher auf die Operationen und die vorher erklärten Membership-Funktionen ein, da diese in Kapitel 4 noch benötigt werden.

Der Vereinigung zweier Funktionen wird der Max-Operator zugesprochen. Der Max-Operator benötigt 2 oder mehrere Funktionen, damit anschließend das in Abbildung 2 dargestellte Ergebnis resultiert.

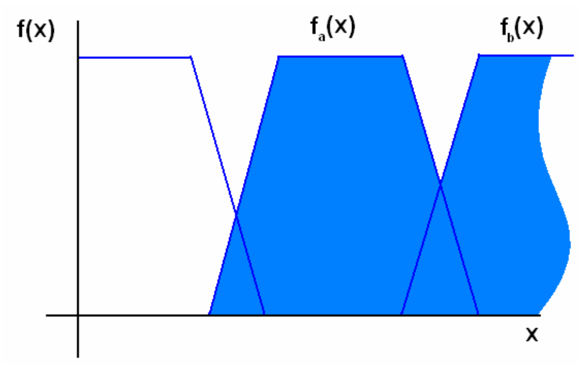


Abb. 2: Max-Operator

Wir gehen in Abbildung 2 davon aus, dass folgende Berechnung durchgeführt wurde:

$$f_C(x) = Max[f_A(x), f_B(x)]$$

Man kann dies Vergleichen mit dem Bool'schen Oder-Operator, welcher auch die Vereinigung zweier Mengen behandelt.

$$f_C = f_A \vee f_B$$

Der Durchschnitt wird auch als Min-Operator bezeichnet, da er den Minimalwert zurückliefert, wie später noch in einem Beispiel näher gebracht wird:

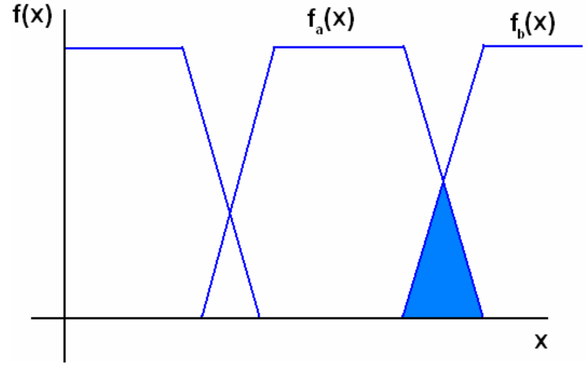


Abb. 3: Min-Operator

In Abbildung 3 wurden die beiden Mengen mit dem Min-Operator zusammengefasst. Auch hier kann daher das Ergebnis mit Hilfe der Formel beschrieben werden:

$$f_C(x) = Min[f_A(x), f_B(x)]$$

Dies ist mit dem logischen Und-Operator gleichzusetzen, welcher auch den Durchschnitt zweier Mengen repräsentiert.

$$f_C = f_A \wedge f_B$$

Neben diesen Definitionen und Begriffen gibt es noch weitere in [2], welche aber in diesem Paper nicht weiter erläutert werden müssen.

2.2. Computing with Words

Computing with Words baut auf Fuzzy-Logik, also auch auf den im vorherigen Kapitel erklärten Methoden auf. Daher werden bei Computing with Words Mengen vereinigt und Namen zugeteilt. Nennen wir diese Menge g und das diese Menge beschreibende Wort w. Diese Darstellung wurde bereits in der Fuzzy-Logik unter den Fuzzy-Sets eingeführt. Dabei kann das Wort atomar sein (z.B. jung) oder einen Verbund darstellen (z.B. nicht jung). Diese Menge g spiegelt die Bedeutung des Wortes w wieder.

Als Beispiel soll folgender Sachverhalt erwähnt werden: „Das Auto fährt schnell." Hier stellt „schnell" das Wort dar, welches eine granulare Menge beschreibt. Das Wort „schnell" beinhaltet somit auch gleichzeitig die Bedeutung der Menge. Man könnte auch sagen: „Das Auto fährt M1." Dabei wüsste man nicht, was M1 bedeutet. Erst durch einen Rückschluss aufgrund der beschriebenen Menge, welche M1 darstellt, kann gefolgert werden, dass dies schnell bedeutet. „Schnell" wird dabei auch noch als fuzzy constraint bezeichnet, da dies für weitere Berechnungen als eine Bedingung angesehen werden kann.

Ein weiteres Beispiel: Wir haben 2 Theoreme (fuzzy propositions p) p1=*Martin wohnt in der Nähe von Klaus*, p2=*Klaus wohnt in der Nähe von Peter*. Wären diese beiden Ausgangssituationen Grundlage für eine Berechnung der Entfernung zwischen Carol und Pat, so müsste man wie folgt vorgehen: Es gibt hier eine granulare Einteilung der Entfernung zweier Personen. Unter Anderem gibt es eine granulare Menge = „in der Nähe von". Diese Menge könnte die Nachbarn in einer Entfernung von 1km beschreiben.

Man kann nun aber nicht direkt daraus schließen, dass aus den beiden Sätzen von oben auch Folgendes resultiert: "Martin wohnt in der Nähe von Peter." Dies hängt von den gesetzten Bedingungen ab. Es ist Festlegungssache, ob die Berechnung „in der Nähe" x „in der Nähe" wieder „in der Nähe" oder „in größerer Reichweite" ergibt. Man kann nun zwar aufgrund der Relativierung der Abstände im 1. Schritt keine konkreten Werte mehr über die Entfernung von Martin und Peter liefern, jedoch stellt dies eher das menschliche Denken dar, als die Rechnung mit konkreten Werten.

Wichtig bei Fuzzy-Logik ist die Begrenzung der Variablenwerte durch die Bedingungen (z.B. 0-1km = *in der Nähe*, 2-5km = *in der Umgebung*, >5km = *weit weg*). In diesem Schritt wird also das Universum an Werten in kleinen Mengen unterteilt und natürlichsprachlichen Wörtern zugeteilt. Dies hat zur Folge, dass Fuzzy-Logik durch die Einführung von Fuzzy-Bedingungen in den Fuzzy-Sets mehr zum menschlichen Denken rückt als die Bool'sche Logik, welches ein Rechnen mit Wörtern ermöglicht.

Aufgrund dieser Granulation gibt es 2 wichtige Notwendigkeiten, welche für die Anwendung von Computing with Words gegeben sein müssen:

- Die Anwendung muss einen bestimmten Grad von Unschärfe zulassen

- Die Anwendung hat als Ziel aufgrund dieser verschwommenen Logik, die Lenkbarkeit, Robustheit und die Nähe zur Realität zu verbessern.

Das System muss also zuerst mit bestimmten Variablen, welche für das Modell von Relevanz sind, beschrieben werden. Als nächster Schritt wird in einem bestimmten Wertebereich eine Granulation vorgenommen. Jedem Bereich g wird dann ein Wort zugewiesen - die Fuzzy-Bedingungen. Die Vorkenntnis des Systems sind somit die Theoreme, von welchen man in der Realität ausgeht (IDS-Initial Data Set). Diese Theoreme stellen mehrere Sachverhalte dar.

Jedes Theorem kann in der Form „X is R" vorkommen. Dabei ist *X* die Variable und *R* die dazugehörige Relation. Diese Eigenschaften stehen in der sog. ED (Explanatory Database). Dabei gibt es noch keine Zusammengehörigkeit der einzelnen Werte, es gibt also noch keine Relation. Erst in der EDI (ED instantiated) sind die Verbindungen vorhanden. Man kann also sagen, dass die ED mehrere mögliche Welten beschreibt, hingegen bei einer EDI schon die genaue Welt beschrieben wird.

Wenn man nun also sagt: „Peter ist jung", so besteht dieses Theorem aus mehreren Relationen, welche in der EDI zu finden sind:

$$ED = Bev\ddot{o}l\ker ung[Peter; Alter] + Jung[Alter, \mu]$$

Hierbei wird zuerst aus der Datenbank das Alter von Peter ermittelt. Anschließend kann mithilfe der Membership-Funktion von „jung" errechnet werden, wie stark Peters Alter der Funktion „jung" zugehört. Diese Zugehörigkeit wird durch μ ausgedrückt.

$$X = Alter(Peter) =_{Alter} Bevölk.[Name = Peter]$$

X wird das Alter von Peter zugewiesen. Alter(Peter) wandelt dabei das konkrete Alter von Peter in eine Fuzzy-Bedingungen (also jung) um.

Es exisiteren jedoch auch Theoreme, die von anderen abhängen. Diese haben die Form:

Wenn x ist R, dann y ist S

Die vorher besprochenen Theoreme haben allerdings alle eine bestimmte Verbindung. Wir haben gesagt: Das Auto fährt schnell und meinen dabei: Die Geschwindigkeit ist schnell. Wir meinen also die Form „X is Y". Im Allgemeinen spricht man aber von einer „X isr Y" Verbindung, da diese von unterschiedlichen Bedingungen abhängen kann. Für r kann anschließend eine der folgenden Funktionen eingesetzt werden:

Zeichen	Bedeutung
e	equal
d	disjunctive
c	konjunctive
p	probabilistic
λ	probabilistic value
u	usuality

Abb. 4: Verbindungsformen bei Theoremen

Im Folgenden werden einige Verbindungen aus Abbildung 4 anhand von Beispielen genauer erläutert:

John spricht Englisch, Französisch und Deutsch.

Proficiency(John) **isc** (1/Englisch + 0,7/Französisch + 0,6/Deutsch)

Hierbei deuten die Zahlen darauf hin, wie gut John die jeweilige Sprache beherrscht.

Eine andere mögliche Form der Bedingungen lautet: „X isp R". Dies bezeichnet eine Verbindung mit einer Normalverteilung mit Mittelwert und Varianz. Man schreibt daher auch:

X **isp** N(m, σ^2)

Hierbei werden der Mittelwert sowie die Abweichung der Normalverteilung beachtet.

Gleiches gilt somit für den Ausdruck:

X **isp** R(0,2/a + 0,5/b + 0,3/c)

Dies bedeutet nun, dass X eine Zufallsvariable darstellt, welche die Werte a, b und c mit den dazugehörigen Wahrscheinlichkeitswerten annimmt.

Wenn man sagt x isu r, dann bedeutet dies „x is usually R" und meint damit,

Prob(X is R) **is usually**.

Wie man also nun erkennen kann, leitet man die Wahrscheinlichkeit eines Theorems mit den dazugehörenden Bedingungen, von dieser Verknüpfung ab. Die meisten Theoreme sind disjunktiver Art. Es gibt nun noch eine Menge Regeln für die Ableitung von Informationen aus anderen. Diese werden für die Arbeit nicht weiter benötigt, können aber in [1] begutachtet werden.

3. Anwendungen und weitergehende Erklärungen

Es wurde nun eine Motivation geliefert und die Grundbegriffe erklärt. Zum besseren Verständnis und für eine tiefer gehende Auseinandersetzung werden jetzt einige Anwendungsgebiete von Fuzzy-Logik erläutert.

Es gibt viele Anwendungsgebiete, bei welchen man mit direkten mathematischen Modellen rechnen kann. Dies ist zumeist dann der Fall, wenn der Mensch selbst bereits diese Normierungen eingeführt hat, und auch Sensoren (Geschwindigkeitsmessgerät, Thermometer usw.) sowie Aktoren (Motoren, Relais usw.) besitzt, welche direkt mit Zahlen arbeiten können. Ein Beispiel bei dem man mit konkreten Zahlen rechnet, wäre die Wettervorhersage. In diesem Feld entstehen genügend konkrete Werte. So wird mit Fakten wie Windgeschwindigkeit in km/h, Hoch und Tiefdruckgebiete in mbar, Luftfeuchtigkeit in Prozent die Vorhersage für die weiteren Tage getroffen.

In vielen anderen Bereichen gibt es keine derartig genauen Messdaten, welche man einem System zuführen kann. Anwendungsbereiche sind Psychologie, Wirtschaft… Hier muss man meist von nur ungenauen Beschreibungen der aktuellen Lage ausgehen. Ungenau beschreibt hierbei ein Zustand, bei welchem die Werte in einem Bereich liegen, und nicht konkret angegeben werden können (siehe granulare Menge). Um trotz dieser natürlichsprachlichen Beschreibung des Systems zu mathematischen Modellen zu gelangen, bedient man sich eben der verschwommenen Logik (Fuzzy-Logik).

Daher lautet nun die erste Aufgabenstellung bei der Anwendung der zuvor dargestellten Methode Computing with Words: Die natürlichsprachige Beschreibung

muss in ein mathematisches Modell übergeleitet werden.

In AI wird dazu oft ein Artificial-Expert-System gebaut. Dieses System liegt jedoch dem menschlichen Denken zugrunde. Die gleichen Funktionen, die ein Experte vollziehen würde, wenn er das Problem hätte, werden in einen Algorithmus implementiert. Der Knowledge Engineer hat dabei die Aufgabe, das System mit Hilfe eines Experten auf der If_Then-Logik zu erstellen. Dabei werden Regeln erstellt, welche dann im Weiteren auf einem Computer implemetiert werden können. Fuzzy-Logik stellt somit eine Methode dar, um effizient verbale Sachverhalte zu verarbeiten.

Diese Schritte sollen nun im nächsten Kapitel durch direkten Einsatz erläutert werden. Im Kapitel 3.1. wird zunächst der Erhalt der Membership-Funktionen näher erklärt. Wenn man dies durchgeführt hat, so muss man für das System Regeln erstellen, was in Kapitel 3.2. durchgenommen wird. In Kapitel 3.3. wird der gesamte Vorgang nochmals für ein einfaches Simulationssystem aufgezeigt.

3.1. Datengewinnung / Granulation

Nun gilt es das Problem zu lösen, dass eine große Datenflut mit konkreten Werten in Fuzzy-Sets automatisch mittels Algorithmus umgewandelt wird. Diese Anforderungen stammen dabei aus der Natur, welche eine hohe Anzahl an Werten zur Verarbeitung liefert. Es müssen passende Mengen gebildet werden, um das System wahrheitsgetreu abbilden zu können. Daher ist es eine wichtige Aufgabe, einen effizienten Algorithmus zu finden, welcher aus einer großen Datenmenge die wesentlichen Informationen extrahiert, welche dann wiederum das IDS für unser Fuzzy-System darstellen [5]. Dabei muss auch darauf geachtet werden, dass die semantik Gap, zwischen der Information in der Datenbank und jener die der Benutzer versteht, geschlossen wird. Wenn es keinen Unterschied bezüglich der Auffassung der Fuzzy-Bedingungen gibt, so sind jene Regeln, mit denen das System arbeitet, semantisch ident zu denen, was der Mensch implizit denkt.

Diese Granulation erfolgt im Wesentlichen in zwei Schritten:

1. Zuerst wird der n-Dimensionale Raum in n-Dimensionale Cluster geteilt. Anschließend wird jeder Raum auf alle Achsen projiziert um einen Zuständigkeitsbereich zur jeweiligen Variable zu erhalten.
2. Die so entstandenen Bereiche können nochmals geclustert werden um eine eindimensionale granulare Menge zu erhalten.

Zuletzt werden diese Punkte noch einer Membership-Funktion zugeordnet. Der vollständige Vorgang kann aus Abbildung 5 entnommen werden.

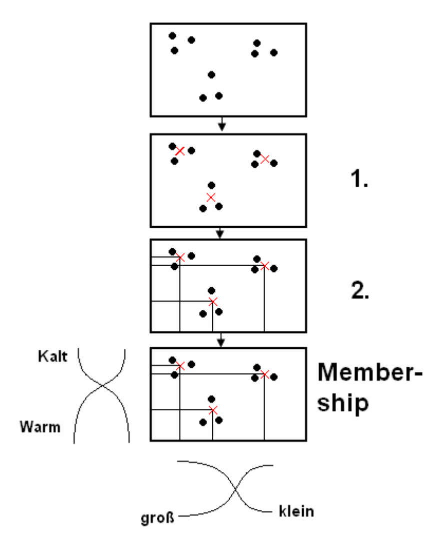


Abb. 5: Vorgang der Datengewinnung

3.2. Ableitung von Fuzzy-Regeln anhand des Beispiels medizinischer Diagnose

Wie im vorherigen Kapitel gezeigt wurde, kann man aus einer Menge von Punkten zuerst in zwei Schritten granulare Mengen bilden und anschließend diesen Mengen Membership-Funktionen zuweisen. Um nun aber mit Wörtern rechnen zu können benötigt es drei Schritte:

1. Die Umwandlung konkreter Werte in granularen Mengen, sowie die Zuweisung zu Bezeichnungen
2. Verarbeitung dieser Wörter
3. Rückwandlung des Ergebnisses auf konkrete Werte

Schritt 1 wurde bereits erklärt. Dieser bildet den Ausgang für den 2. Schritt. Die Abfolge wird in diesem Kapitel anhand der Medizin näher erläutert. Der letzte Punkt wird bei Bremssteuerung (Kapitel 4) näher dargestellt.

Ein wichtiger Anwendungsfall von Computing with Words ist die medizinische Diagnose [11]. Hierbei handelt es sich einerseits um einen stark zahlenlastigen Input (Temperatur, Herzfrequenz, Blutwerte…), jedoch wird für die Diagnose selbst mit Wörtern gerechnet. Dies liegt einfach im Denken des Menschen. Der Arzt denkt sich: Der Patient hat hohe Temperatur, die Blutwerte sind in Ordnung, der Blutdruck ist gering usw. Daher muss der Patient an einer bestimmten Erkrankung leiden. Wie im 1. Schritt erklärt, wird dieses n-Dimensionale Fuzzy-Set granularisiert, sodass nun Mengen vorhanden sind, welche alle ein, in Hinsicht auf diese Variable, homogenes Verhalten aufweißen. Wie stark die Ausprägungen dieses Verhaltens sind, drückt die Membership-Funktion aus. Es ist nun also der nächste Schritt, geeignete Regeln für die Erkennung der Krankheit zu bilden. Die Regeln hierbei haben alle die Form:

$$Wenn[Symptom_1(\mu_1) \wedge Symptom_2(\mu_2)]$$
$$Then[Diagnose(d, \mu]$$

Um diese Regeln ableiten zu können, benötigt man das Expertenwissen. Somit wurden die Regeln gebildet, und das System erfolgreich erstellt.

3.3. Die Natur besser verstehen

Es gibt eine starke Tendenz in Fuzzy-Modellierung, dass man zuerst die Natur beobachtet, diese natürlichsprachlich beschreibt, weiters mit Fuzzy-Logik in ein mathematisches System umwandelt und anschließend dieses Modell bei Maschinen einsetzt. Zum besseren Verständnis soll ein Ameisenmodell beschrieben werden, welches zwar keinen typischen Anwendungsfall von Computing with Words darstellt, aber trotzdem die Arbeitsweise verdeutlicht. Es handelt sich dabei um eine Gruppenproblemlösung (Cooperative Problem Solving), bei welcher die Ameisen in Kolonien eine hohe Intelligenz zu Tage bringen [3] [6].

Ameisen und auch andere Tiere müssen oftmals bei ihrer Futtersucher einen weiten Weg zurücklegen. Daher bedienen sie sich eines Mechanismus, welcher ihnen erlaubt, in kürzester Zeit den für sie besten Weg herauszufinden. Dies geschieht über die Kennzeichnung der Pfade. Je öfter dabei ein Weg begangen wurde, desto passender scheint dieser Weg zu sein. Nun kann man die Behauptung aufstellen, dass wenn die Ameisen den schlechteren Weg gehen, diesen ja auch markiert haben, und daher beide Wege gleichermaßen verwenden. Es ist jedoch der schlechtere Weg jener, bei dem die Ameisen länger benötigen, um zur Futterstelle zu gelangen. Dieser wird dann natürlich häufiger frequentiert, was zu einer größeren Kennzeichnung durch Pheromone führt. Weiters verschwindet diese Kennzeichnung auch mit der Zeit, sodass der schlechte Weg alsbald eine niedrigere Priorität aufweißt. Dies passiert so lange, bis fast jede Ameise den kürzeren Weg benutzt. Diese Methode der Natur wurde näher in [4] erläutert.

Nun existieren, wie bereits besprochen, viele mögliche Gründe, warum man ein natürliches Phänomen in ein mathematisches Modell abbilden will (Simulation, Parameterbeeinflussung, Vorhersagen usw.). Daher müssen als erster Schritt die Variablen deklariert werden. Da der gewählte Weg (TDS) von der Pheromondichte abhängt, muss als einzige Variable hierbei die Pheromonkonzentration auf den einzelnen Wegen erstellt werden.

Als nächstes muss das Verhalten, also die Regeln der Ameisen, näher betrachtet werden. Wir gehen im Weiteren von lediglich zwei Wegen aus. Die Ameise wählt jenen Weg, welcher mehr Pheromone aufweißt. Wenn L subtrahiert von R größer 0 ergibt, geht sie nach Rechts, ansonsten nach Links. L und R sind dabei die Pheromonkonzentrationen auf den beiden Wegen.

Als 3. Schritt werden die Zugehörigkeitsfunktionen zu den granularen Mengen, wie sie in Punkt 2 erstellt wurden, gebildet. Da es nur eine Variable gibt, können wir auch nur ein Diagramm erstellen, welches in Abbildung 6 dargestellt wird.

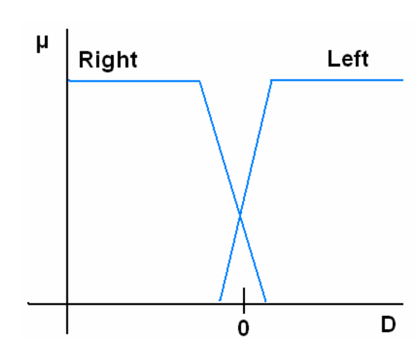


Abb. 6: Membership-Funktion für Ameisenwahl

Dieses Diagramm ist auch stark davon abhängig, wie man das „Expertenwissen“ der Ameise modellieren will. Knapp bei 0 weiß die Ameise, dass es auch der andere Weg sein kann, welcher schneller zum Futter führt. Sie wendet aber trotzdem den Weg mit der größeren Pheromonkonzentration an.

Als letzten Punkt muss man noch ein Auswerteverfahren wählen. Da man nun alles modelliert hat, um eine Fuzzy-Lösung zu erhalten, muss das Ergebnis wieder in eine konkrete Antwort umgeleitet werden. Dies kann mittels unterschiedlicher Verfahren gesche-

hen. Die Verfahren werden genauer in Kapitel 4 erklärt.

Die meisten Bereiche der Natur werden mit einer verbalen Beschreibung des Sachverhaltes abgebildet. Die Wissenschaft kann stark davon profitieren, wenn sie diese in mathematische Modelle umwandeln kann. Dazu kann und wird auch Fuzzy-Logik verwendet.

Dieser Trend ist vor allem deswegen nützlich, um natürliche Mechanismen, welche sich schon in der realen Welt bewährt haben, nachzuahmen. Diese Wissenschaft wird Biomimicry genannt. Dazu wird die reale Welt durch die natürliche Sprache beschrieben. Mit der Fuzzy-Logik kann diese in ein mathematisches Modell übergeleitet werden, welches dann einfach auf Maschinen angewendet werden kann.

3.4. Objektorientiertes Modellieren mit Wörtern

Dieses Kapitel zeigt einen Anwendungsfall auf, welcher zwei Methoden verknüpft. Einerseits die objektorientierten Paradigmen und andererseits die Prinzipien von Fuzzy-Logik [7].

Der Nachteil von objektorientiertem Modellieren ist, dass ein Objekt genau einer Klasse zugehört. Dies kann durch Fuzzy-Logik umgangen werden, da man hier nicht nur sagen kann ob, oder ob ein Element nicht einer Klasse zugehört, sondern auch, in welchem Ausmaß. Ein Nachteil von Fuzzy-Logik ist dadurch gegeben, dass man stets nur Aussagen über bestimmte Eigenschaften tätigen kann (z.B.: Das Auto fährt schnell) und nicht über vollständige Klassifikationen.

Als ein Tool für objektorientiertes Modellieren kann Frill++ herangezogen werden. Dies ermöglicht es, an vielen Stellen mit Wörtern zu rechnen. Zuerst sei die Unschärfe bei den Eigenschaften von Objekten erwähnt. Es ist hier erlaubt, den konkreten Objekten unscharfe Attributwerte zuzuweisen. Weiters erlaubt dieses Schema eine “Fuzzy-Vererbung”. So kann ein Objekt durch Vererbung mehreren Klassen angehören. Jedoch wird hierbei nicht alles vererbt, sondern nur spezielle Attribute und Methoden. Ein weiterer Punkt ist das Zuweisen von Attributen und Methoden, welche anschließend für die Zuordnung eines Objektes zu einer Klasse herangezogen werden.

4. Vollständiges Beispiel: Bremssystem

Um nun den gesamten erläuterten Stoff der vorherigen Kapitel in einem durchgängigen Praxisbeispiel zu erläutern, soll ein Bremssystem für ein Auto erstellt werden. Weiters stellt dies ein Beispiel dar, bei welchem Computing with Words wohl heutzutage am meisten eingesetzt wird, nämlich bei Fuzzy-Reglern.

4.1. Ausgangssituation

Ein Auto fährt hinter einem anderen. Ständig werden die Geschwindigkeit des vorderen Autos sowie der Abstand zum vorderen Auto gemessen (vgl. Abbildung 7). Nun soll ein Bremssystem erstellt werden, welches eben aufgrund dieser beiden Variablen, einen konkreten Bremswert ermittelt. Weiters soll wegen der einfacheren Bauweiße ein Fuzzy-Regler zum Einsatz kommen.

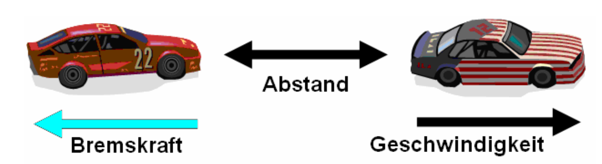


Abb. 7: Ausgangssituation

4.2. Erstellung des Systems

Wie in Kapitel 3 beschrieben wurde, müssen zuerst Terme definiert werden, welche die drei Variablen in granulare Mengen spalten. Diese können der Einfachheit halber in drei Mengen eingeteilt werden:

Abstand	klein	Geschw.	klein	Bremsen	schwach
	mittel		mittel		mittel
	groß		hoch		stark

Als nächstes wird eine Fuzzyfizierung vorgenommen. Das bedeutet, dass die konkreten Werte durch eine Membership-Funktion, zu den einzelnen granularen Mengen zugewiesen werden. Für nähere Einzelheiten siehe Anhang Abbildung 8, 9 und 10.
Hat man nun die Mengen gebildet, so müssen mit diesen Sets Regeln aufgestellt werden, welche man von Experten in diesem Bereich einholen muss (Abb. 11).

4.3. Inferenz

Es tritt jedoch häufig der Fall ein, dass mehrere Regeln zur Verwendung kommen, sodass man einen Mechanismus benötigt, welcher die Membership-Werte (μ) für die Bremskraft berechnet. Zur Lösung des Problems kann man auf mehrere Methoden zurückgreifen. Meistens wird jedoch die Max-Min Inferenz verwendet.

Hierbei behandelt man die Werte für die Therme untereinander mit dem Min-Operator, anschließend das Ergebnis für die gleiche Gruppe mit dem Max-Operator.

Weiters soll auch noch die Max-Prod Inferenz erklärt werden. Hierbei werden erneut zuerst die Terme mit dem Min-Operator verknüpft, anschließend jedoch wird der Ausgangswert mit dem minimalen Erfülltheitsgrad multipliziert. Für nähere Erklärung siehe [8].

Als nächsten Schritt können also konkrete Werte mit Hilfe der Membership-Funktionen zu den granularen Mengen zugewiesen werden. Aufgrund der Regeln wird dann ein Fuzzy-Ergebnis für die Bremskraft abgelesen. Jedoch muss auch dieser unscharfe Wert wieder umgerechnet werden. Dabei gibt es mehrere Auswertmethoden, welche im nächsten Unterkapitel erklärt werden.

4.4. Defuzzifizierung

Es stehen viele Methoden zur Verfügung, um die Werte wieder zurück zu rechnen, denn das Bremssystem kann nicht mit dem Wert *0,8 stark* bremsen

COA bedeutet Center of Array. Dazu bindet man die Ergebnisse wieder in die Graphik ein. Der Mittelpunkt der sich ergebenden Fläche bildet den gesuchten Bremswert.

MoM (Mean of Maximum) verwendet hingegen den vereinfachten Algorithmus. Hierbei wird lediglich die Mitte des Maximums verwendet (siehe Abbildung 12).

Neben diesen Methoden gibt es noch CoM (Center of Maximum) und andere. Weitergehende Informationen sind in [13] zu finden.

4.5. Ergebnis: Bremsvorgang

Es soll nun für die folgende Situation die Bremskraft ermittelt werden: Die Geschwindigkeit beträgt 45km/h und der Abstand 30m.

Fuzzyfiziert ergibt für 45km/h einen Wert von klein{0,5} mittel{0,5} und groß{0}. Dieselbe Prozedur muss für den Abstand durchgeführt werden: klein{0,5} mittel{0,25} und hoch{0} (laut Abb. 8 und 9). Bildet man aufgrund des Expertenwissen die Bremswirkung ergeben sich mehrere Ergebnisse für *mittel* (Abb. 11).

Die Regel lautet nach Max-Min Inferenz: Die Terme selbst müssen mit dem Min-Operator behandelt werden. Die Ergebnisse in derselben Gruppe müssen untereinander mit dem Max-Operator verknüpft werden. Also überprüfen wir zuerst *klein* x *klein*. Dies ergibt die Bremswirkung *stark*. *0,5* MIN *0,5* ergibt *0,5*. Also haben wir zuerst stark=*0,5*. Diesen Vorgang wiederholen wir für alle Kombinationen, sodass wir als Resultat für die Bremskraft mittel=*0,25* und *0,5* erhalten. Aufgrund des Max-Operators muss hier *0,5* gewählt werden. Die Endwerte für die Bremswirkung lauten somit: schwach{0,25} mittel{0,5} und stark{0,5}. Mit diesem Ergebnis kann jedoch noch kein Bremssystem arbeiten. Aufgrund dessen, muss dies wieder in einen realen Wert umgewandelt werden. Dies wird bei uns mittels CoA durchgeführt. Wenn man den Schwerpunkt des Diagramms auswertet, so kommt man ungefähr auf 5,5 m/s² (siehe Anhang Abb. 12).

Eine Simulationssoftware mit dem Namen Suco-SOFT fuzzyTECH kann von der Adresse www.moeller.net/de bezogen werden (Abb. 13, 14 und 15).

5. Fazit & Ausblick

In diesem Paper wurde zuerst auf die grundlegenden Dinge von Fuzzy-Logik eingegangen. Diese mussten, da Computing with Words ein Einsatzgebiet von Fuzzy-Logik ist, näher erläutert werden. Hierbei wurde auch erklärt, warum seit Beginn von Fuzzy-Logik, deren Wichtigkeit immer weiter steigt. Anschließend konnte auf Grundsätzliches bei Computing with Words näher eingegangen werden. Später wurden einige Anwendungen näher gebracht. Dies begann mit dem direkten Einsatz von Computing with Words für das bessere Verständnis von Ameisen. Weiters konnte eine neue Sichtweise auf objektorientiertes Modellieren gezeigt werden. Durch den Einsatz von Fuzzy-Logik in diesem Bereich, kommt es zu neuen Möglichkeiten, welche es mit Java in dieser Form nicht gibt. Java weist, im Gegensatz zu Frill++, Objekte fix einer Klasse zu. Abschließend wurde die Konstruktion eines Computing with Words Systems anhand der medizinischen Diagnose dargestellt. Wie wir gemerkt haben, benötigt diese jedoch viele Schritte. Um nun trotzdem den gesamten Verlauf, sowie die dargestellten Definitionen zu verstehen, bedienten wir uns eines anschaulichen Beispieles.

Computing with Words lässt also neben Forschungszwecken sowie den direkten praktischen Einsatz auch noch Platz für neue Sichtweisen, weshalb es seit der Einführung von L.A. Zadeh stets an Anwendern der Methode zunimmt. Neueste Forschungen gehen sogar den noch weiteren Weg, indem sie das Unscharfe weiter unscharf machen wollen. Man will die Unschärfe des menschlichen Handelns noch besser einbinden. Dies funktioniert einerseits mit Einführung von Unschärfen bei den Regeln und andererseits durch eine neue Art von Fuzzy-Reglern [12]. Man kann also gespannt sein, was dieses Thema an neuen Methoden aufwerfen wird.

6. References

[1] L. A. Zadeh, "Fuzzy Logic = Computing with Words", *Life Fellow*, IEEE Transactions on Fuzzy Systems, University of California, 2. Mai 1996

[2] L. A. Zadeh, "Fuzzy Sets", *Information and Control8,* National Science Foundation, University of California, 1965

[3] V. Rozin, M. Margliot, „The Fuzzy Ant", *IEEE Computational Intelligence Magazin,* School of Electrical Engeneering Systems, Tel Aviv, 2004

[4] S. Goss, S. Aron, J. L. Deneubourg, and J. M. Pasteels, Self-organized shortcuts in the Argentine ant," *Naturwissenschaften, vol. 76*, 1989, pp. 579-581

[5] G. Castellano, A. M. Fanelli, C. Mencar, Generation of interpretable fuzzy granules by a double-clustering technique, *Archives of Control Science, Special Issue on Granular Computing*, University of Bari, 2002

[6] E. Tron, M. Margliot, Mathematical Modeling of Observed Natural Behavior: A Fuzzy Logic Approach, *Fuzzy Sets and Systems, vol. 146,* University Tel Aviv, 2002

[7] J.M. Rossiter, T.H. Cao, T. P. Martin, J. F. Baldwin, Object-Oriented modelling with words, *Fuzzy Systems, The 10th IEEE International Conference on,* University of Bristol, 2001

[8] H. Kiendl, Fuzzy-Control methodenorientiert, Oldenbourg, München Wien 1997, ISBN 3-486-23554-0

[9] L. A. Zadeh, Knowledge Representation in Fuzzy Logic, *Transactions on Knowledge and Data Engeneering,* IEEE, 1989

[10] L.A. Zadeh, Commonsense Knowledge Representation Based on Fuzzy Logic, *Computer, vol.16,* IEEE, University of California, October 1983, pp. 61-65

[11] G. Castellano, A. M. Fanelli, C. Mencar, Discovering Human Understandable fuzzy diagnostic rules from medical Data, *Proc of the third European Symposium on Intelligent Technologies (EUNITE 2003),* University of Bari, 2003

[12] T. Ozen, J. M. Garibaldi, Effect of Type 2 Fuzzy Membership Function Shape in Modelling Variation in Human Decision Makin, Automated Scheduling, *Fuzzy Systems, Proc. 2004 IEEE International Conference on,* University of Nottingham 2004

[13] M. Gerke, M Grof, Seminar Fuzzy-Logik, Fachgebiet Elektrotechnik, FernUniversität Hagen, 2001

ANHANG

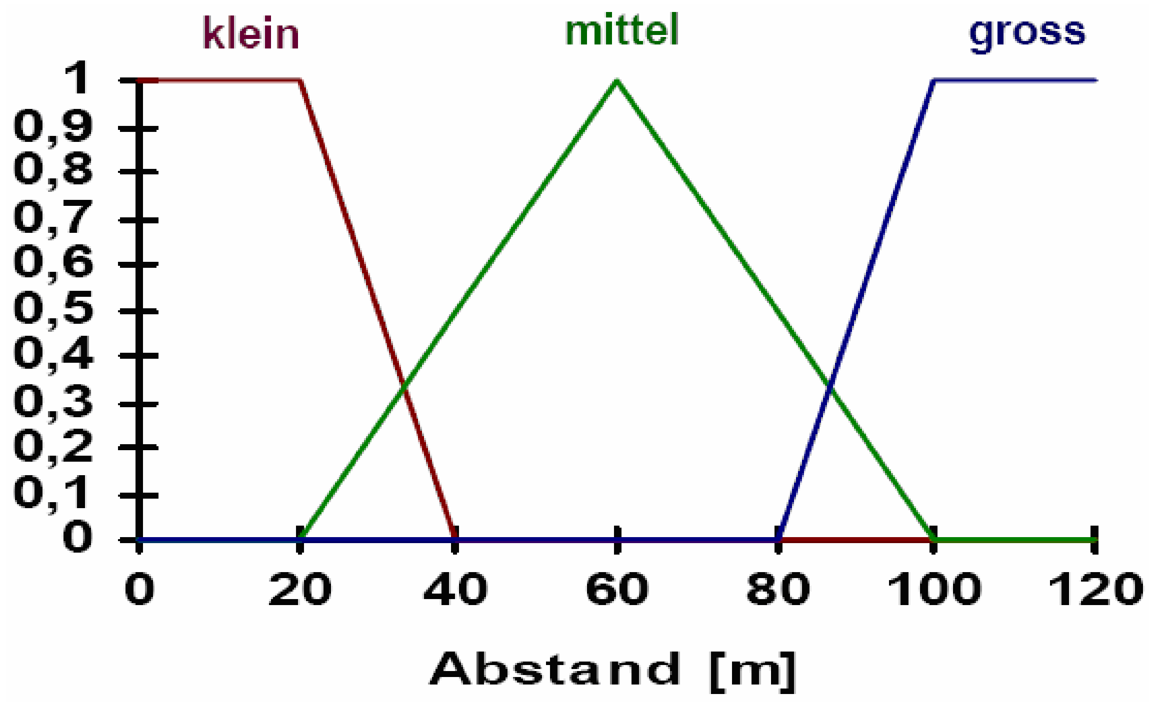


Abb. 8: Fuzzyfizierung der Variable „Abstand“

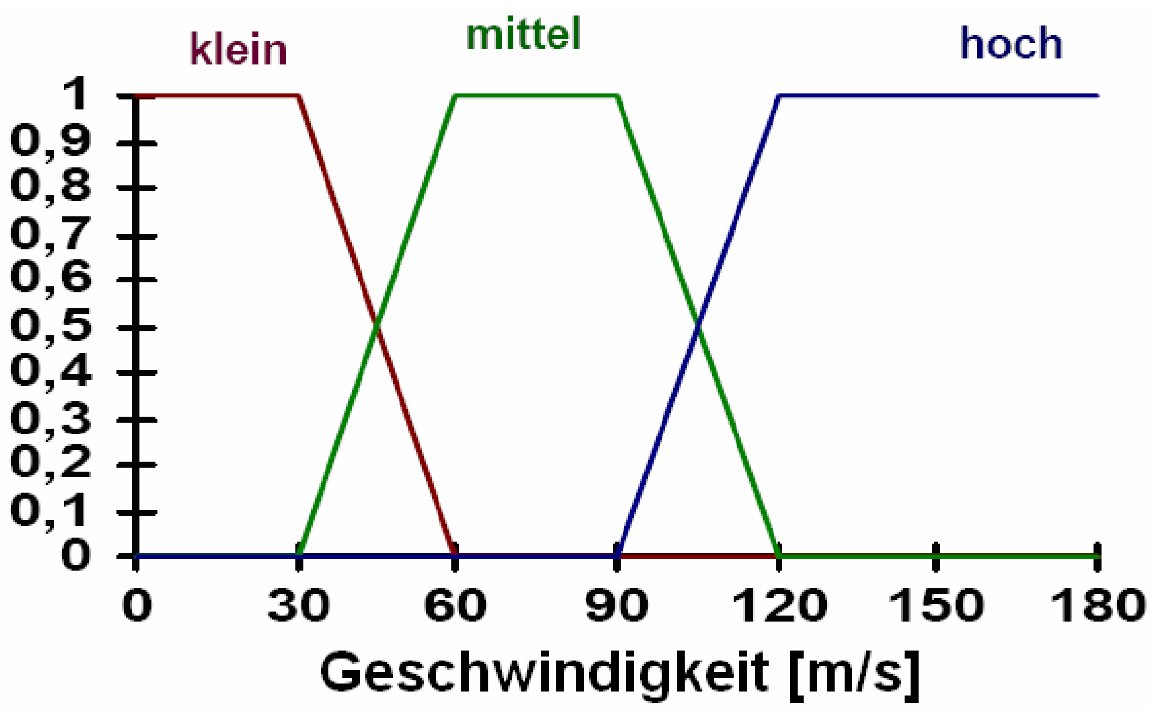


Abb. 9: Fuzzyfizierung der Variable „Geschwindigkeit“

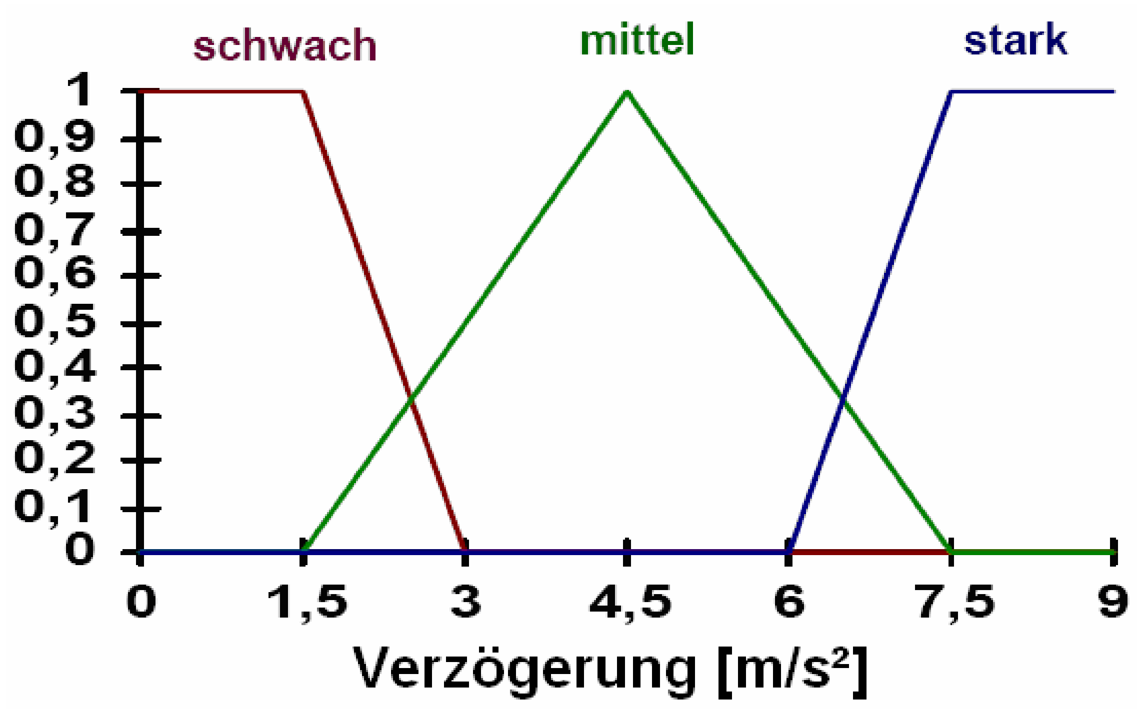


Abb. 10: Fuzzyfizierung der Variable „Bremskraft“

Regelbasis

Regel 1	WENN	Abstand =klein	UND	Geschw. = klein	DANN	Bremsen = stark
Regel 2	WENN	Abstand =mittel	UND	Geschw. = klein	DANN	Bremsen = mittel
Regel 3	WENN	Abstand =groß	UND	Geschw. = klein	DANN	Bremsen = schwach
Regel 4	WENN	Abstand =klein	UND	Geschw. =mittel	DANN	Bremsen = mittel
Regel 5	WENN	Abstand =mittel	UND	Geschw. =mittel	DANN	Bremsen = schwach
Regel 6	WENN	Abstand =groß	UND	Geschw. =mittel	DANN	Bremsen = schwach
Regel 7	WENN	Abstand =klein	UND	Geschw. =hoch	DANN	Bremsen = schwach
Regel 8	WENN	Abstand =mittel	UND	Geschw. =hoch	DANN	Bremsen = schwach
Regel 9	WENN	Abstand =groß	UND	Geschw. =hoch	DANN	Bremsen = schwach

Abb. 11: Regelbasis für das Bremssystem laut Experten

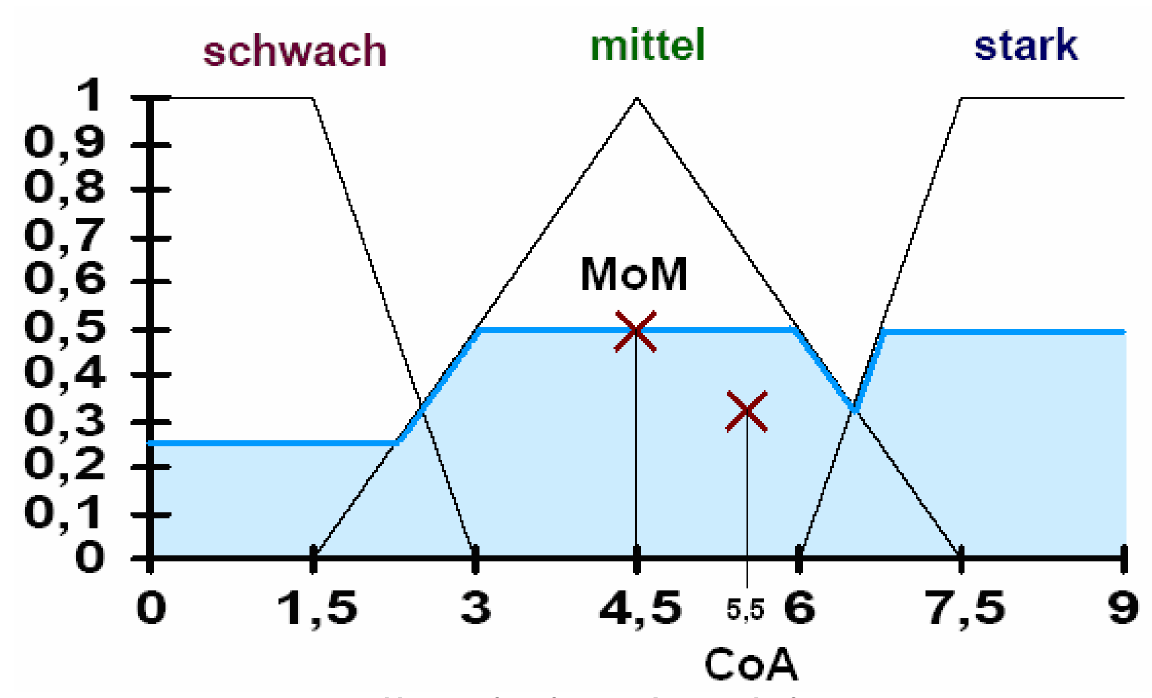


Abb. 12: Defuzzyfizierung der Bremskraft

Bilder der Simulationssoftware:

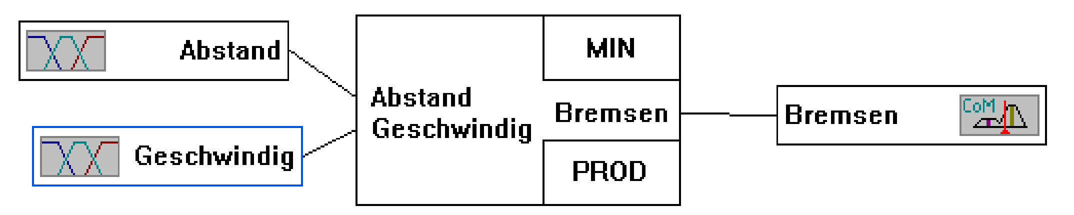
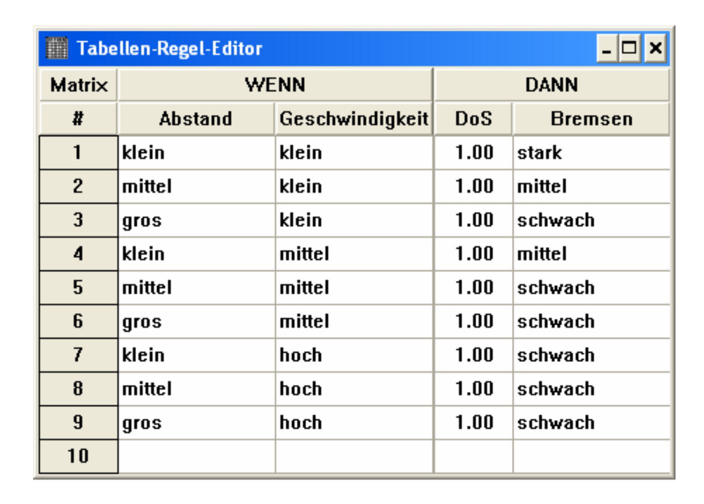


Abb. 13: Modellierung des Bremssystems in SuccoSoft



Tabellen-Regel-Editor

Matrix	WENN		DANN	
#	Abstand	Geschwindigkeit	DoS	Bremsen
1	klein	klein	1.00	stark
2	mittel	klein	1.00	mittel
3	gros	klein	1.00	schwach
4	klein	mittel	1.00	mittel
5	mittel	mittel	1.00	schwach
6	gros	mittel	1.00	schwach
7	klein	hoch	1.00	schwach
8	mittel	hoch	1.00	schwach
9	gros	hoch	1.00	schwach
10				

Abb. 14: Regelbasis in SuccoSoft

Ergebnis:

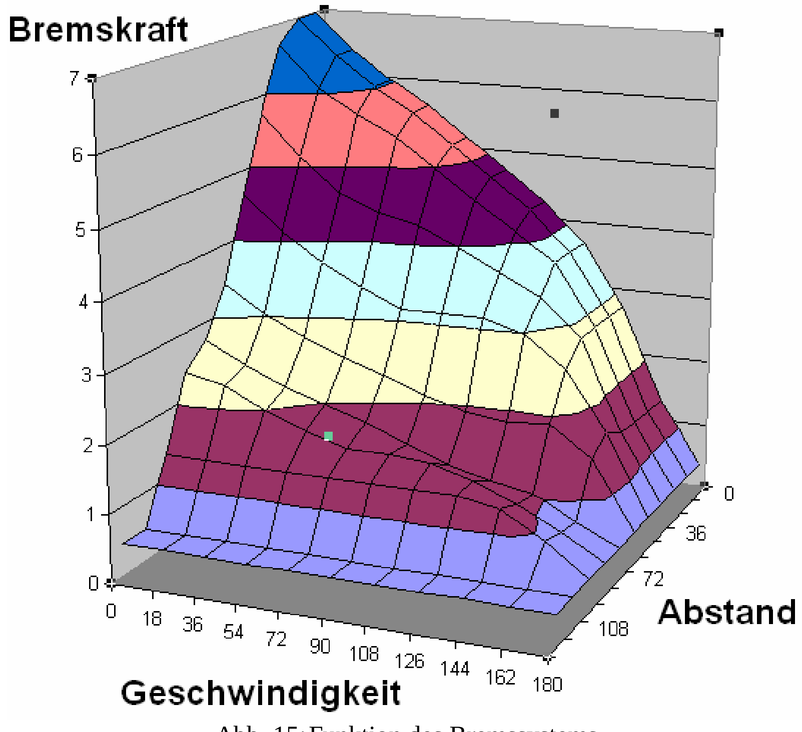


Abb. 15: Funktion des Bremssystems